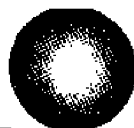
目　录

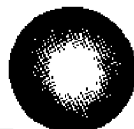


第二篇 系 统 篇

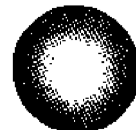




第三篇 应 用 篇







第一篇 基 础 篇

第1章 嵌入式Linux基础

知识点：

- 嵌入式系统的应用
- 嵌入式系统的特点
- 实时与实时系统
- 嵌入式 Linux 的特点与现状
- 各种主流嵌入式微处理芯片

本章导读：

本章将主要介绍嵌入式 Linux 的基本知识。通过本章的学习，读者可以对嵌入式系统的应用场合、特点及对实时与实时操作系统的基本特点有所了解，为学习后续章节的内容打下良好的基础。



1.1 嵌入式系统

1.1.1 嵌入式系统的应用

当前，人类进入信息大爆炸的时代，各类信息极度丰富，数字信息技术和网络技术高度发达，只有借助各种计算机，才能对各类信息进行处理。同时，这些计算机不再局限于以前的 PC，而是包括形态各异、性能千差万别的各类嵌入式系统——从基于群集的超级计算机到嵌入在冰箱中的微控制器。

后 PC 时代的到来，使得人们开始越来越多地接触到一个新的概念——嵌入式产品。嵌入式产品遍布于人们的日常生活，从手机、PDA 到家中的空调、冰箱，从小汽车到波音飞机，甚至武器库中的巡航导弹。数字时代的标志不再是一台一台的 PC，而是形态各异的嵌入式系统。

嵌入式系统的概念的提出已经有相当长的时间，其历史几乎和计算机的历史一样长。但在以前，它主要用于军事领域和工业控制领域，所以很少被常人关注和了解。直到最近，随着数字技术的发展和新的体积更小的控制芯片和功能更强的操作系统的出现，它才被广泛应用于人们的日常生活中。

现在，嵌入式产品已经在很多领域得到广泛的使用，如国防、工业控制、通信、办公自动化和消费电子领域等。

1. 工业过程控制

工业过程控制即是对工业生产过程中的生产流程加以控制。这种控制是建立在对被控对象和环境不断进行监控的基础上的。在控制过程中，嵌入式的计算机处于中心位置，它通过分布在工业生产中的各个传感器收集信息，并对这些信息进行加工处理和判断，然后向执行器件发出控制指令。整个控制流程如图 1-1 所示。

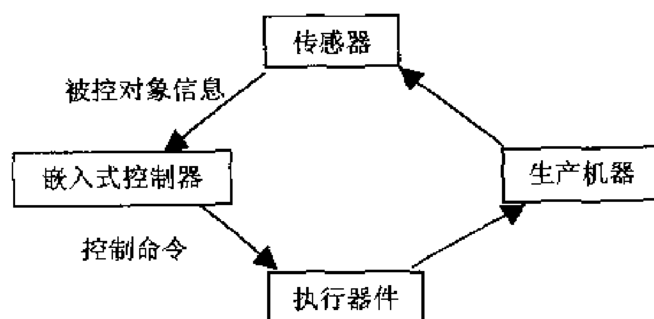


图 1-1 工业控制中的嵌入式应用模型

2. 军事电子设备和现代武器

这是早期嵌入式系统的重要应用领域，军事领域从来就是许多高新技术的发源地。由

于内装嵌入式计算机的设备反应速度快，自动化程度高，所以威力巨大，自然很得军方青睐。从爱国者导弹的制导系统到战斗机的瞄准器，从 M1A2 的火控系统到单兵系统的通信器，都可觅得嵌入式系统的踪迹。

3. 网络通信设备

众多网络设备都是嵌入式系统使用的典型例子，如路由器、交换机、Web 服务器、网络接入设备等。另外，在后 PC 时代将会产生比 PC 时代多成百上千倍的瘦服务器和超级嵌入式瘦服务器，这些瘦服务器将为人们提供需要的各种信息，并通过 Internet 自动、实时、方便、简单地提供给需要这些信息的对象。设计和制造嵌入式瘦服务器、嵌入式网关和嵌入式因特网路由器已成为嵌入式系统的一大应用方向，这些设备为企业信息化提供了廉价的解决方案。

4. 消费电子产品

后 PC 时代的消费电子产品应具有强大的网络和多媒体处理能力。易用的界面和丰富的应用功能。这些特性的实现，都依赖于嵌入式系统提供的强大的数字处理能力和简洁实用的特性。

作为移动计算设备的 PDA 和手机已出现融合趋势，未来必然是二者合一，提供给用户随时随地访问 Internet 的能力，同时还具有其他信息服务功能，如文字处理、邮件管理、个人事务管理和多媒体信息服务等。而且简单易用，价格低廉，维护简便。

作为消费电子产品的嵌入式系统的另一大应用是信息家电。信息电器是指所有能提供信息服务或通过网络系统交互信息的消费类电子产品。如前几年打得火热的“维纳斯”与“女娲”之战就是信息家电中的机顶盒之争。如果在家电中的冰箱、空调、监控器等设备中嵌入计算机并提供网络访问能力，用户就可以通过网络随时随地地了解家中的情况，并控制家中的相应电器。

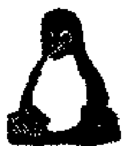
总之，随着信息技术的发展，人类进入一个全新的数字时代，数字化产品空前繁荣，嵌入式系统被应用于空前广泛的领域。在以后相当长一段时间内，嵌入式技术将在消费电子领域进一步飞速发展，嵌入式产品将与人们的生活结合得越来越紧密。

1.1.2 嵌入式系统的特点

什么叫嵌入式系统？嵌入式系统就是以应用为中心，以计算机技术为基础，软硬件可裁减，适合应用系统对功能、可靠性、成本、体积和功耗要求的专用的计算机系统。

在嵌入式系统中，计算机系统一般作为智能控制部件嵌入到整个应用系统中，是整个系统的控制中心，主要用于对系统的信息处理部件和用户交互界面加以控制。在这种情况下，用户并不知道（或者不需要知道）嵌入的计算机的存在，系统控制软件一般被固化在嵌入式计算机中，嵌入式计算机一般不需要（或不能）被用户重新编程，通过特殊的输入、输出设备与系统进行交互。

任何嵌入式系统都包括硬件和软件两个方面。硬件包括微处理器、存储器、I/O 端口和



图形控制器等。软件包括操作系统软件和应用软件，应用软件控制着嵌入式系统的运作和行为，而操作系统则为应用程序提供必要的底层支持，它一般是通过提供应用编程接口（API）来实现的。但在嵌入式系统开发中它们的区别又不是绝对明显的，有时嵌入式系统的开发者可能要做操作系统和应用编程两方面的工作。

因为嵌入式系统是面向应用、产品和用户的，所以不可能不研究应用特性以开发出一个如 PC 般通用的嵌入式系统。在嵌入式系统中，具体的应用将决定对硬件和软件的需求，如芯片、存储器、I/O 扩展和操作系统、应用程序编制等。和通用计算机不同，嵌入式系统的硬件和软件都必须高效率地设计，量体裁衣，去除冗余，尽量以最小的系统、最低的成本去实现目标功能，这样的产品才具有竞争力。它通常都具有低功耗、体积小、集成度高等特点，能够把通用 CPU 中许多由板卡完成的任务集成在芯片内部，从而有利于嵌入式系统设计趋于小型化，移动能力大大增强，跟网络的结合也越来越紧密。

下面具体介绍嵌入式系统的软硬件特性。

1. 硬件特性

嵌入式系统总是面向特定应用的，与通用 PC 的硬件相比，它的硬件系统具有以下特性：

- 体积小，集成效率高。嵌入式系统总是去除冗余，力争用最小的系统完成目标功能，特别在一些手持设备中更是这样。
- 面向特定应用的特性。具体嵌入式系统只能适合某一特定应用，针对另一应用就需要重新设计硬件系统。
- 低功耗，电磁兼容性好，能在恶劣环境下工作，即使死机也要求能够快速重启。

总之，嵌入式系统硬件在价格、功能、体积、重量、能耗等方面都有严格的限制。

2. 软件特性

软件是一个应用系统的灵魂，对于嵌入式系统的软件部分，它具有以下特点：

- 嵌入式软件的研发与硬件紧密相关。由于嵌入式软件的开发是针对具体硬件平台进行的，它往往牵涉硬件驱动方面的一些软硬结合部分，这就要求开发人员必须具备相关的硬件知识。
- 软件代码要求高效率和高可靠性（小）。由于嵌入式系统中软件运行空间有限，内存空间非常宝贵，在软件的编程过程中必须时刻考虑软件的运行效率，同时选用高质量的编译工具。在实时系统中，处理器必须严格处理异步发生的各种任务，这对程序的算法设计提出了更高的要求。另外，嵌入式软件系统还应该有异常处理、快速复位等特点。
- 软件一般固化在 FLASH 或 ROM 中。为了提高执行速度和系统的可靠性，同时缩短系统复位时间，一般在嵌入式软件调试好后，会下载固化到目标板中的 FLASH 或 ROM 中。目标板启动时，再运行其中的代码，而不是像 PC 那样从硬盘存储器中读取程序。

1.2 实时与实时系统

1.2.1 实时与实时系统的基本概念

在嵌入式领域中一个非常重要的概念是实时，实时系统是指在确定的时间内完成规定功能，并能对外部异步事件作出正确响应的计算机系统。设计实时系统时，不仅要考虑算法逻辑设计的正确性，而且要考虑执行这些算法的时间相关性。

实时系统的核心是必须在确定的时间内执行完一项预先定义的操作，否则将引起性能下降甚至系统崩溃等严重后果。比如在一个工业控制系统中，传感器每秒钟收集 5 组数据，而处理器必须根据每组数据发出相应的控制指令，这就要求处理器处理每组数据的时间不能大于 1/5 秒，否则在上一组数据还没有处理完，相应的控制命令不能及时发送出去，新的一组数据又输入进来，这样必将导致系统崩溃。

需要说明的是，实时系统并不是说系统的响应和处理速度非常快；而一个高速系统也不一定是实时系统，就如一个使用 UINX 操作系统的大型服务器，不一定是实时系统一样。实时系统对处理速度的要求是千差万别的，必须视具体应用而定。高性能的粒子甄别系统的甄别处理可能要求在微秒甚至纳秒内作出判断，而上一个例子只要求秒级的处理速度。

实时系统中实时性的实现需要硬软件的配合来完成。首先必须保证硬件的处理速度满足实时要求，同时相对软件而言，实时性体现在组成软件系统的各个任务的执行时限。在实时系统中，每个任务的运行都是有时限（deadline）的，即任务执行的最长时间。

实时系统按时限对系统的影响程度不同，分为软实时系统（soft real-time system）和硬实时系统（hard real-time system）。前者对偶尔超过时限的操作是可以容忍的，并且具有相应的处理机制，不会因为个别存在的超时限操作而引起严重的后果，尽管这可能造成系统性能的下降；而后者是不能容忍的，在硬实时系统中，超过时限的后果是无法预测的。

在实时系统中，用响应时间（response time）、生存时间（survival time）和吞吐量（throughput）等 3 个指标来衡量系统的实时性。下面对这 3 个指标进行相应的说明。

- 响应时间：是实时系统从识别一个外部事件到作出响应的时间，在控制应用中它是最重要的指标，如果在硬实时系统中外部事件不能及时响应处理，系统将可能崩溃。
- 生存时间：是数据的有效等待时间，在该时间内数据是有效的，超过生存时间的数据是无效的，如在上面的工业控制的例子中，数据的生存时间为 1/5 秒，超过这段时间缓冲区的数据就将无效。
- 吞吐量：是在一个给定时间内，系统可以处理的事件总数。例如 MODEM 中用每秒钟处理的比特数表示吞吐量。吞吐量可能是平均响应时间的倒数，但一般要小一些，因为在每次响应后还要做处理工作，有一段恢复时间。

与 PC 中所使用的操作系统一样，在实时领域也有实时操作系统。实时操作系统（Real-time Operating System，RTOS）是基于计算机的，是管理计算机软硬件资源并提供



人机命令和编程接口的系统，它能在固定的时间内对一个或多个由外设发出的信号作出适当的反应，同时它还是嵌入式软件开发平台。与普通分时操作系统不同，实时系统强调系统对外部异步事件响应时间的确定性。RTOS 是操作系统的一个重要分支，它与通用的商用操作系统如 UNIX、Windows 有共同的一面，也有不同的一面。对于一般的商用操作系统，它的目的是方使用户管理计算机资源，以最大限度地利用计算机的资源；而 RTOS 追求的是任务切换的实时性、事件响应时间的可确定性及系统的高可靠性。评价一个实时系统的性能要从任务调度功能、内存管理功能、最小内存开销、任务切换时间、最大中断禁止时间等方面来考虑。下面对这些概念进行简单的说明，详细介绍将在后面章节中进行。

1. 任务与任务调度

在 RTOS 中的任务是指一个程序分段，这个分段被操作系统当作一个基本单位来调度，它大致相当于 PC 机中操作系统的进程的概念。任务是一个可运行的程序段，它一般完成一项单一的功能。任务自己认为它独自占有 CPU 资源。在嵌入式软件的设计过程中，要把问题分化为一个一个的任务，每个任务都是应用的某一部分，完成一部分功能，它们被赋予一定优先级，有自己的一套 CPU 寄存器和堆栈空间。在嵌入式软件设计中多任务的存在可以使 CPU 利用达到最大值，并且可以实现程序模块化。

在只有一个 CPU 的嵌入式系统中，多任务运行的实现要靠 RTOS 在许多任务之间切换、调度。任务调度即决定应该运行哪个任务，这是内核的主要职责之一。多数实时操作系统都是基于优先级调度算法的。即每个任务都被赋予一定的优先级，而 CPU 总是执行处于就绪态的优先级最高的任务，但究竟何时让优先级最高的任务掌握 CPU 使用权，要看 RTOS 的内核类型（即是可剥夺抢占式的，还是不可剥夺抢占式的），具体相关知识的介绍，请参见第 3 章关于任务调度的内容。

2. 内存管理

它是针对有内存管理单元（MMU）的处理器设计的一些桌面操作系统，如 Windows、Linux，都引入了虚拟存储器的概念。虚拟内存地址被送到内存管理单元（Memory Management Unit，MMU）。在这里，虚拟地址被映射为物理地址，实际存储器被分割为相同大小的页面，采用分页的方式载入进程。一个程序在运行之前，没有必要全部装入内存，而是仅将那些当前需要运行的部分页面装入内存运行。在许多 RTOS 中，也采用 MMU 进行内存管理，并且它的内存管理模式可以分为实模式和保护模式。

还有许多嵌入式系统是针对没有 MMU 的处理器设计的，在这里不能使用处理器的虚拟内存管理技术，采用的是实存储器管理策略。因而对于内存的访问是直接的，它对地址的访问不需要经过 MMU，而是直接送到地址线上输出，所有程序中访问的地址都是实际的物理地址。而且，大多数的嵌入式操作系统都对内存空间没有保护，各个进程实际上都共享一个运行空间。一个进程在执行前，系统必须为它分配足够的连续地址空间，然后将其全部载入主存储器的连续空间，如嵌入式 Linux 领域的 uClinux 就是这样。

在嵌入式系统的开发中，RTOS 的内存管理能力是选择实时操作系统的重要参考因素，因为嵌入式开发人员也不得不参与系统的内存管理。从编译内核开始，开发人员必须告诉

系统这块开发板到底拥有多少内存；在开发应用程序时，必须考虑内存的分配情况并关注应用程序需要运行空间的大小。另外，若采用实存储器管理策略，用户程序同内核以及其他用户程序都处在一个地址空间中，程序开发时要保证不侵犯其他程序的地址空间，以使得程序不至于破坏系统的正常工作，或导致其他程序的运行异常。

3. 最小内存开销

在 RTOS 评价中，最小内存开销是一个重要指标。最小内存开销是指在嵌入式系统中只运行某个操作系统所需要的最小内存。在嵌入式系统中，由于内存有限，所以必须考虑运行操作所需的最小内存。系统内存必须大于最小内存开销，因为运行应用软件还需内存。

4. 任务切换时间

由于某种原因使某个任务退出运行时，RTOS 要保存它的现场运行信息，把它插入相应任务队列，并根据一定的算法重新选定一个任务使之投入运行，这一过程所需的时间称为任务切换时间。任务切换时间也就是操作系统从一个任务取回 CPU 控制权把它交给另一个任务所需要的时间。

5. 中断禁止时间与中断延迟时间

当 RTOS 运行在核心态或执行某些 API 时，是不会因为外部中断的到来而中断执行的，只有在重回用户态后，才会响应中断。从系统进入核心态到重回用户态的最大时间就称为中断禁止时间。

中断延迟时间是指从系统确认中断到开始执行中断服务程序的第一条指令的这一段时间。RTOS 的中断延迟时间由以下 3 个因素决定：

- 处理器硬件延迟时间，也就是从外部电路产生中断到形成中断请求信号的时间，这段时间一般非常短，可以忽略。
- RTOS 处理中断并将控制权移交给相应的中断处理程序所需时间。
- 中断禁止时间。

为了缩短中断延迟时间，许多商用的 RTOS 采用"可中断"的内核调用，即使在内核态，也可以中断执行，响应外部中断。

任务切换时间和中断禁止时间决定了系统的实时性。以上几个方面是 RTOS 性能优劣的重要指标，在具体选择时应依据应用要求和成本综合考虑。

1.2.2 目前应用广泛的嵌入式实时操作系统

其实，嵌入式系统并不是一个新生的事物，从 20 世纪 80 年代起，国际上就有一些 IT 组织和公司开始进行商用嵌入式系统和专用操作系统的研发，这其中涌现了一些著名的嵌入式系统。进入 20 世纪 90 年代后，国内的一些公司和科研院校也开始涉足嵌入式实时操作系统，并且已经开发出了一些可与国外产品媲美的实时操作系统。下面对一些在我国广泛使用的 RTOS 进行相应的介绍。



1. VxWorks 操作系统

VxWorks 是目前嵌入式系统领域中使用最广泛、市场占有率最高的嵌入式实时操作系统。它是美国 WindRiver 公司的产品，以其良好的可靠性和卓越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中，已经在包括爱国者导弹和火星探测器等许多领域上得到成功应用。它具有以下特性：

- 微内核结构（最小体积<8KB）。
- 微秒级中断处理。
- 高效的任务管理：
  - 多任务，最多具有 256 优先级。
  - 优先抢占和轮转调度。
  - 快速、确定的上下文转换。
- 多处理器支持。
- 灵活的任务间通信。支持以下多种任务间通信方式：
  - 具有优先级继承的二进制、计数器、互斥的信号量。
  - 消息队列。
  - 套接字。
  - 共享内存。
  - 信号异常处理。
- 符合 POSIX 1003.1b 实时扩展标准。
- 支持 MS-DOS 和 RT-11 文件系统。
- 完全符合 ANSI C 标准。
- 支持多种体系结构的处理器，如 x86、i960、Sun Sparc、Motorola MC68xxx、ARM、POWER PC 等。

2. WinCE 操作系统

WinCE 是由微软公司推出的嵌入式实时操作系统，Microsoft Windows CE 是从整体上为有限资源的平台设计的多任务、多优先级的操作系统。其模块化设计允许它对于从掌上电脑到专用的工业控制器的用户电子设备进行定制。操作系统的基本内核需要至少 200KB。但它的最大缺点是实时性不好，是软实时操作系统，只能用于对实时性要求不高的场合。

3. VRTX 操作系统

VRTX 是老牌的嵌入式实时操作系统，在早期的商用嵌入式操作系统中曾扮演“领头羊”的角色。

VRTX 具有一组满足用户需要的模块化的编程界面和工具。无论是对存储器和耗能有限制的手持器件，还是用于电话交换的网络管理卡，VRTX 都有一套工具满足开发者的需求。通过对可向上兼容编程接口和面向对象开发的支持，VRTX 保证在将来支持和能重复利用其源码。VRTX 的特点包括易于载入定制硬件、文件系统支持及 ANSI/POSIX 接口。

VRTX 还支持多种网络协议。

VRTX 目前主要有 VRTXsa、VRTX5 和专门支持单芯片（SOC）的 VRTXmc/VRTXoc 3 个版本。

4. pSOS 操作系统

ISI 公司推出了 RTOS，现在 ISI 已经被 WinRiver 公司兼并，pSOS 属于 WindRiver 公司的产品。这个系统是一个模块化、高性能的实时操作系统，专为嵌入式微处理器设计，提供一个完全多任务环境，在定制的或是商业化的硬件上提供高性能和高可靠性。可以让开发者根据操作系统的功能和内存需求定制成每一个应用所需的系统。开发者可以利用它来实现从简单的单个独立设备到复杂的、网络化的多处理器系统。

5. Palm OS 操作系统

3Com 公司的 Palm OS 在 PDA 市场上占有很大的市场份额，它有开放的操作系统应用程序接口，开发商可以根据需要自行开发所需要的应用程序。

6. Delta OS 操作系统

Delta OS 是电子科技大学实时系统实验室和北京科银京成公司联合开发的，国内具有完全自主知识产权的 RTOS，已经成功地应用于消费电子产品、通信产品、工业控制及军用电子产品中。它具有高可靠性、实时性良好、可伸缩性、可移植性强（90%以上代码采用 C 语言编写）、支持多处理器结构的硬件环境和接口标准的开放性等特点。

Delta OS 是由具有高可靠性和实时性的内核 DeltaCOR、嵌入式 TCP/IP DeltaNET、嵌入式文件系统 DeltaFILE 以及嵌入式图形接口 DeltaGUI 组成。

7. 嵌入式 Linux 操作系统

以上介绍的 RTOS 都是商用的嵌入式操作系统，它们在系统可靠性和对用户的技术支持上都有自己的优势。但是，这些专用操作系统均属于商业化产品，其价格昂贵；而且，由于很多时候它们的核心源代码都是不公开的，这使得每个系统上的应用软件与其他系统都无法兼容。由于这种封闭性还导致商业嵌入式系统在对各种设备的支持方面存在很大的问题，使得对它们的软件移植变得很困难。由于 Linux 自身诸多优势，在嵌入式这个 IT 产业的新的关键领域，嵌入式 Linux 操作系统适时地出现在各嵌入式厂商面前，吸引了许多开发商的目光，成为嵌入式操作系统的新宠，下面将对它进行重点介绍。

1.3 嵌入式 Linux

1.3.1 从 Linux 到嵌入式 Linux

Linux 是一种在网络上产生的操作系统，它的诞生已有 10 来年的历史，最初源自芬兰学生 Linus Torvalds 学习操作系统课程后的练习。1991 年 9 月 17 日，在实现了 Linux 最初



的基本功能后，Linus 将 Linux 以开放源代码的方式放在网络上（这就是 Linux 0.01 版本），吸引了一大批顶级黑客加入到 Linux 的开发队伍中，使得 Linux 在短期内就成为一个稳定、成熟、实用的操作系统。

与传统的操作系统不同，Linux 操作系统的开发一开始就在 FSF（自由软件基金会组织）的 GPL（GNU Public License）的版本控制之下，Linux 内核的所有源代码都采取了开放源代码的方式。其内核的发布是由 Linus Torvalds 和 Alan Cox 等领导的内核开发小组控制。世界各地的开发者们都将自己对 Linux 内核所作的修改提交给 Linus 小组，由这个小组进行统一控制，随时对内核进行更新升级。整个开发过程都使用 CVS（著名的自由版本控制软件）进行版本控制，使得全世界的几千名开发者可以通过网络进行协同开发。Linux 内核开发的速度相当快，目前在其站点（http://www.kernel.org）上几乎每间隔三天就进行一次内核的升级，目前最新的稳定内核是 2.6.20。

和其他操作系统相比，Linux 操作系统具有以下优点：

（1）内核稳定、功能强大、支持多种硬件平台、应用软件多、兼容性好。

Linux 是一个类 UNIX 的操作系统，其代码是完全重新开发的，内核功能强大，实现简洁。它提供了类 UNIX 的编程接口和系统调用，可以方便地将 UNIX 系统上的应用程序，移植到 Linux 上运行。另外由于它开放源代码的缘故，使得 Linux 的源代码得到了众多网络黑客的审查和修改，有着非常好的稳定性，可以胜任 7（天）×24（小时）、365（天）×24（小时）等高稳定性应用程序的要求。

最新的 Linux 内核支持多种体系结构的处理器，包括目前流行的 Intel x86、Motorola/IBM PowerPC、ARM、Compaq Alpha、Sun SPARC 等微处理器体系结构。目前，很多将 Linux 移植到其他硬件平台的工作也都正在进行之中。

同时，Linux 平台上的应用软件也不断地得到扩充。许多著名的商业软件都有了 Linux 下的版本。像办公自动化的 Star Office、文字处理、电子表格软件 Corel WordPerfect 8、集成开发环境 Kdevelop、数据库 Oracle 8 for Linux、Netscape Navigator 6.0 浏览器、Apache 网络服务器等重量级应用程序都已经纷纷推出。而且，目前像 Borland 这样的顶级商业 RAD 开发工具厂商，也推出了其旗舰产品 Delphi 的 Linux 版本 Kylix，使得在 Windows 平台下使用 CLX 开发的应用程序可以方便地移植到 Linux 环境，简化了 Linux 下应用软件的开发。Kylix 开发速度相当快，目前最新版本已经是 Kylix 3。另外，像 KDE、GNOME 等强大的桌面环境的推出，在很大程度上也提高了 Linux 的易用性。

（2）内核可根据需要任意裁减。

根据 GPL 协议规定，Linux 操作系统的源代码可以免费获得，并在遵守 GPL 协议的条件下进行修改。这就使得修改 Linux 内核来满足自己的需要成为可能。目前，众多的嵌入式 Linux 版本，正是受益于 Linux 的这一特点。

（3）使用成本低。

使用商业操作系统（如 Windows、Solaris、AIX 等）都需要为每一个复制支付相当数量的费用，同时在其下的应用软件也都需要购买才能使用。为了在商用操作系统下建立一个开发环境，除了要为操作系统本身付费之外，还要为组成开发环境的应用软件支付大量费用。相比之下，Linux 是免费软件，只需要遵守 GPL 的规定，就可以免费获得复制。Linux

下有同样遵循 GPL 规定的汇编（如 PASCAL、C、C++、Java 等）的软件工具开发包，如大名鼎鼎的 GNU 工具链（如 GCC、GDB 等）。这些软件从功能角度上看并不亚于商用开发包，甚至优于某些商业产品，如 GCC、Apache 等。这样，使用 Linux 构建服务器系统或是软件开发环境等，其软件购买费用几乎可以忽略不计。

（4）多专业的商业公司参与，发展潜力大

Linux 开放源代码的特点，使得一大批专门商业公司都参与到对 Linux 的开发之中，如 RedHat、VALinux 和 CYGWIN 等。这些公司一方面参与 Linux 的开发，一方面将 Linux 内核和应用软件整合起来，加入自己的安装和配置管理工具打包发行，这就是常说的 Linux 发行版本。Linux 的发行版本数量众多，最有名的 Linux 发行版本有 RedHat、Slackware、Mandalaka、Debian、SESU 等。这些 Linux 发行版本整合了大量的 Linux 应用软件，并且提供了相对容易的使用和管理界面，极大降低了 Linux 的使用难度。

综上所述，Linux 具有相当多的优点。它的内核稳定、功能强大，可裁减和低成本的特性非常适合嵌入式应用。但是，Linux 最初并不是为嵌入式系统设计的。Linux 内核本身不具备强实时特性，且内核体积较大，因此，想要把 Linux 用于嵌入式系统，必须对 Linux 进行实时化、嵌入式化，而这正是目前嵌入式开发的新热点，也是本书所要介绍的重点。

嵌入式 Linux 就是在嵌入式系统中使用的 Linux。通常是将标准 Linux 进行相应改造后，再用作嵌入式计算机的操作系统，为嵌入式应用程序提供操作系统服务。

1.3.2 嵌入式 Linux 的特点

在本章 1.1 节已经提到，嵌入式系统所具有的特殊要求包括嵌入式系统有不同程度的实时性要求；嵌入式系统在体积、功能、能耗等方面受具体工作环境和开发、生产成本的限制；嵌入式系统的软件硬件环境复杂多变，嵌入式系统的操作系统应该根据这些环境有很好的可移植性、可配置性和可剪裁性，以便能灵活地适应不同的软硬件环境。

要把 Linux 用于嵌入式环境，就必须修改 Linux 满足嵌入式系统的要求。在 1.3.1 小节中讲到，要修改 Linux 使其嵌入式化。主要集中在两个方面：一是体积，二是实时性。针对这两个方面，国外一些公司和研究机构开发出了各具特色的嵌入式 Linux 版本。在实时性方面，最为有名的嵌入式 Linux 版本有 RTLinux 和 KURT_Linux 等。在减小 Linux 体积方面，则以美国网虎国际公司最为突出，已经成功地将 Linux 剪裁到 143KB 大小，这是目前正式公布的最小 Linux 内核。

与目前市场上的众多商业的 RTOS（实时操作系统）相比，嵌入式 Linux 拥有以下的特点。

1. 完全开发源代码

嵌入式 Linux 开放源代码，这使得学习、修改、剪裁 Linux 成为可能，嵌入式系统的设计者可以对嵌入式 Linux 进行二次开发，去掉操作系统的附加功能，只保留必须的操作系统功能，并且可以根据实际应用的需要优化操作系统的代码，从而降低整个系统开销与能耗。而目前大多商用 RTOS，要么不提供源代码，要么购买源代码需要支付高额的版本



费用。相比之下，嵌入式 Linux 的这一优点对于对成本和能耗极为敏感的嵌入式系统是十分重要的。

2. 成本低

GPL 协议保证了源自 Linux 的嵌入式 Linux 也是开放源代码的自由软件，也就是说，只要遵守 GPL 协议，嵌入式 Linux 操作系统的源代码可以自由获得。因此，使用嵌入式 Linux 开发嵌入式应用，用于购买操作系统软件的费用可以忽略不计。而商业的 RTOS，其操作系统的每个拷贝售价在几百美元到几万美元不等，如果需要操作系统的源代码，则还需要另外购买。另外，大多数嵌入式 Linux 使用的开发工具也是遵守 GPL 协议的，同样可以免费获得。

3. 丰富的实用软件支持

Linux 操作系统是一个完整的、功能强大的操作系统、提供了大量的实用程序和各种各样的应用软件。这些软件的正确性和有效性都经过了实际应用检验，可以根据需要，利用 Linux 提供的丰富的软件支持，迅速构建嵌入式应用的软件环境。这样可以极大地减小嵌入式系统软件开发的时间和费用，提高系统的可靠性。而商用的 RTOS 尽管也试图提供各种常用软件工具包支持，但是其数量是无法和 Linux 操作系统匹敌的。

可以看出，嵌入式 Linux 操作系统与传统的商业 RTOS 相比有着自己独特的优点。根据一家专门进行嵌入式 Linux 系统信息发布的网站 http://www.Linuxdevices.com 的调查，有 52%的用户决定在未来 24 个月内使用 Linux 作嵌入式系统的开发原型，而只有 29%的用户仍然使用专有操作系统，19%的用户仍然使用 Windows 系列操作系统进行嵌入式系统开发。这说明了嵌入式 Linux 在开发嵌入式系统的广阔前景。

1.3.3 嵌入式 Linux 发展现状

Linux 的嵌入式改造主要围绕体积和实时性展开，目前已经有许多公司在进行这方面的工作。在本节中，将介绍几款著名的嵌入式Linux 系统，包括 RTLinux、KURT_Linux、uClinux、Xlinux 和 Embedix 等。下面将介绍几款应用广泛的嵌入式 Linux 操作系统。

1. RT-Linux

RT-Linux 是利用 Linux 进行实时系统开发比较早的尝试，美国新墨西哥理工学院计算机系的 Victor Yodaiken 和 Michael Baranov 早在 1996 年就开始了对 RT-Linux 的开发。其最新版本是 RTLinuxPro 1.2 版。目前 RT-Linux 已成功应用于从航天飞机的空间数据采集、科学仪器测控到电影特技图像处理等众多领域。根据 Linuxdevices.com 的调查情况，有 5%的用户已经将 RT-Linux 应用到它们的嵌入式应用中。

RT-Linux 的原理是采用双内核机构，即将 Linux 的内核代码进行少量修改，将 Linux 任务以及 Linux 内核本身作为实时内核的一个优先级最低的任务，而实时任务优先级高于普通 Linux 任务，即在实时任务存在的情况下运行实时任务，否则才运行 Linux 本身的任

务。实时任务不同于 Linux 普通进程，它是以 Linux 的内核模块（Linux Loadable Kernel Module，LKM）的形式存在的。需要运行实时任务的时候，将这个实时任务的内核模块插入到内核中去。实时任务和 Linux 一般进程之间的通信通过共享内存或者 FIFO 通道来实现。RT-Linux 工作原理如图 1-2 所示。

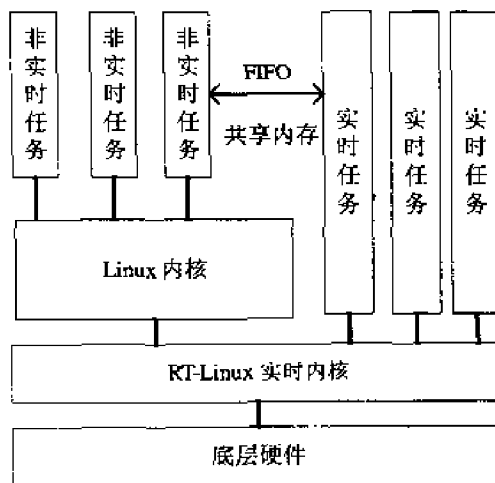


图 1-2 RT-Linux 原理图

图 1-2 中可以看出，RT-Linux 接管了机器的硬件，实时任务直接在 RT-Linux 实时内核的调度下运行，这样保证了任务的实时性。而通过将 Linux 本身作为实时内核优先级最低的任务，又使得 Linux 的常规操作可以正常运行。同时，实时任务和 Linux 普通进程之间可以相互配合，将需要强实时特性的任务放在实时内核内运行，将界面显示和用户监控功能放在 Linux 内部执行。这种实现方法既可以充分利用 Linux 的强大功能，又可以保证关键任务的实时性。

2. KURT_Linux

KURT_Linux 则采用了另一种方式来获得实时性。KURT_Linux 由美国 Kansas 大学研制。研发 KURT_Linux 的最初目的是满足实时网络多媒体处理方面研究的需要。因为 ATM 网络和多媒体处理既要求有很高的实时性，又要求全面的操作系统服务，传统的分时系统和专用实时系统都不能同时满足这两方面的需要，因而决定改造 Linux 来满足要求。通过直接对 Linux 核心进行改造来实现目标，采用的方法比较简洁，没有大动干戈，却基本达到了目的。

KURT_Linux 强化了 Linux 的时钟机制和调度机制。标准 Linux 将时钟间隔固定为 10ms，也就是在最好情况下，Linux 也需要 10ms 才能进行一次重调度，这显然无法满足实时操作的要求。KURT_Linux 通过修改 Linux 的时钟管理模块，使得时钟得以 μs 为单位在任何需要的时候都可以产生中断。这样，既保证了响应的实时性，又避免了不必要的开销。另外 KURT_Linux 通过增加了新的实时调度模块，使得在 KURT_Linux 可以同时调度实时任务和分时任务运行。



3. uClinux

uClinux 则从另一方面来增强嵌入式操作系统的性能。众所周知，虚拟存储器管理是传统通用操作系统的一大特色功能。但是对于嵌入式实时系统来说，则是不实用的。首先，绝大多数的嵌入式设备都没有大容量的硬盘支持，虚拟存储器管理所需的外部存储器将无法获得。其次，虚拟存储器管理将影响系统的实时性能。例如，从储藏柜里找东西吃，找东西是需要时间的，找到之后还要放到盘子里，端到桌子上来。这总是没有直接在桌上的盘子里夹菜来得快。当然，前提是食物不是太多，而桌子也够大，能放得下全部的美味。

uClinux 的基本思想就是去掉标准 Linux 里的虚拟存储器管理功能，虽然强大的存储器管理功能是标准 Linux 的一大优点，这样一方面减小了内核的体积，另一方面又增强了系统的实时性能。uClinux 是基于 Linux 2.0 或者 Linux 2.4 的，目前已经移植到很多的处理器平台上，包括 68k、PowerPC、ARM 等平台上。

另外，在 Linux 小型化方面还有一些版本。像前面提到了美国网虎国际公司推出的号称是目前世界上最小的嵌入式 Linux——Xlinux，其核心只有 143KB，而且还在不断地缩减中。Xlinu 的另一项特点就是“超字元集”技术，它不仅兼容标准 Linux 字符集，还支持 12 个国家和地区的字符集。

而 Embedix 嵌入式 Linux 则是由嵌入式 Linux 行业主要厂商 Lineo 推出的。Lineo 对 Linux 进行重新设计，在较小的体积内，提供了 25 种以上的系统服务，包括像 Web 服务器这样的高级功能。

嵌入式 Linux 版本众多，在此只能涉及沧海一粟，更多的资源可以到网上搜索。

另外，目前关于嵌入式 Linux 实时化和微型化还有一些发展方向，比如嵌入式的文件系统、嵌入式 C 库、嵌入式环境的集成开发工具等，限于篇幅也不再介绍。

1.4 主流嵌入式芯片简介

嵌入式处理器是嵌入式系统中的核心部件。嵌入式处理器的功能和性能，影响着整个系统的设计。嵌入式处理器的选择，制约了其配套的外围器件的选择，也很大程度上影响着系统软硬件功能的划分策略，包括操作系统的选择。嵌入式处理器的功能强弱决定了系统的性能指标的上限。

与通用计算机处理器不一样，由于嵌入式系统的多样性，不可能有一种通用的嵌入式处理器能满足所有嵌入式系统的需要。这迫使人们设计了各种各样的嵌入式处理器来满足不同领域的应用要求。如 51 系列单片机广泛用于工业过程控制，而 Motorola 8260 则主要用于电信和网络市场的通信专用芯片。这样造成嵌入式处理器的种类繁多、数量庞大。据不完全统计，目前全世界嵌入式处理器的品种总量已经超过 1000 种，流行体系结构有三十几个系列。现在几乎每个半导体制造商都在生产嵌入式处理器，越来越多的公司都拥有自己的（嵌入式）处理器设计部门。

嵌入式处理器和通用处理器相比，在价格、功能、体积、重量、能耗等方面都有严格

的限制。因此，嵌入式处理器大多采用整合集成的办法，以增强处理器的竞争力。通常以某一种微处理器内核为核心，在芯片内部集成 ROM/EPROM、RAM、总线、总线逻辑、定时/计数器、看门狗、I/O 控制器、A/D、D/A、FLASH、RAM、EEPROM 等各种必要功能和外设。为适应不同的应用需求，一般一个系列的处理器都具有多种衍生产品，每种衍生产品的处理器内核都是一样的，不同的只是存储器和外设的配置及封装。这样可以使单片机最大限度地和应用需求相匹配，功能正好满足应用需求，从而减少功耗和成本，提高系统的可靠性。这种思想的一个极端就是单芯片系统（SOC，system on chip），即在一个芯片上实现一个完整的系统。

目前，一种比较流行的分类方法，将嵌入式处理器分为 4 类：

- 嵌入式微处理器（Embedded Microprocessor Unit，EMPU）。
- 嵌入式微控制器（Microcontroller Unit，MCU）。
- 嵌入式 DSP 处理器（Embedded Digital Signal Processor，EDSP）。
- 嵌入式片上系统（System On Chip）。

嵌入式微处理器是和通用计算机的微处理器对应而言的。在应用中，一般是将微处理器装配在专门设计的电路板上，在母板上只保留和嵌入式相关的功能即可，这样可以满足嵌入式系统体积小和功耗低的要求。目前的嵌入式处理器主要包括 PowerPC、Motorola 68000、ARM 系列等。

嵌入式微控制器又称为单片机，它将 CPU、存储器（少量的 RAM、ROM 或两者都有）和其他外设封装在同一片集成电路里（如常见的 8051）。

嵌入式 DSP 专门用来对离散时间信号进行极快的处理计算，提高编译效率和执行速度。DSP 正被广泛应用于数字滤波、FFT、谱分析、图像处理的分析等领域。

由于嵌入式 Linux 大多运行在嵌入式微处理器上，在本书中，将主要介绍与嵌入式微处理器相关的内容。下面将介绍目前市面上几家主要嵌入式芯片厂商生产的主流产品。

1.4.1 Motorola 公司嵌入式芯片简介

摩托罗拉公司是全球最大的嵌入式处理器制造商，其嵌入式处理器芯片主要集中于网络和数据通信领域。摩托罗拉的嵌入式处理器芯片主要基于先进的 RISC 结构的 PowerPC 处理器内核，其产品有好几个系列，包括著名的 68 系列、ColdFire 系列、MPC 系列等，品种多达几十种，其型号包括 MC68302、MC68360、MPC850、MPC860、MPC8240、MPC8241、MPC8245、MPC8260、MCF5272、MCF5249、MCF5407、MC68EZ328 等。另外，摩托罗拉公司也生产龙珠芯片，如 MC9328MX1。

68 系列是摩托罗拉公司的早期产品，其型号从最初的 68 到 680，再到 6800，最后到 68000。68 系列的产品，经历了一个相当长的发展过程，其产品成熟、稳定。并且，已经得到广大用户的认可，被广泛地应用于嵌入式系统的各个领域，像工业过程控制、信息家电、单板计算机和通信协议控制处理等。典型的产品如 MC68302，它只是一个集成的多协议处理器（Integrated Multiprotocol Processor，IMP），内部集成了微处理器和一些通信控制



领域的常用外围组件，特别适合于通信行业的应用。

ColdFire 的设计者将处理器定位在 32 位 RISC 基础上，其体系结构建立在以前的 68000 之上，充分考虑向下的兼容性，以保护 68000 系列产品用户的投资。ColdFire 改进了 68000 系列的体系结构，提高了计算性能并增加部分 DSP 的功能。ColdFire 能够支持充足的嵌入外围设备，并且功耗极低，这些优点使其成为一个能处理一系列的消费和商务电器的嵌入式处理器。

MPC850、860、8260 等处理器则为网络和数据通信领域提供了广泛的支持。实际上 MPC 系列也是基于 68000 系列的。如 MPC860Powe4UICC 是 MC68360 在网络和数据通信领域的替代升级产品，它全方位提高了 MC68360 器件的运行性能，包括器件的适应性、扩展能力和集成度等。MPC860Powe4UICC 采用双处理器内核结构，即为高层应用服务提供高性能通用 32 位处理器 PowerPC 内核，为底层通信应用服务提供专用 RISC 处理器内核，两者可通过片内的双端口内存通信，快速高效完成对通信协议的处理。MPC850 则是 MPC860 的简化版本。而 MPC8260 PowerQUICC II 则号称是目前最先进的专为电信和网络市场而设计的集成通信微处理器。它同样采用了双处理器内核结构。其通用 32 位处理器采用高速的、高性能超标量体系结构微处理器 EC603e，其运行速度最高可达 200 MHz；其 32 位 RISC 通信控制器也得到增强，加上 MPC8260 集成了大量的网络和通信外围设备，MPC8260 为用户提供了一个全新的整个系统解决方案来建立高端通信系统。

1.4.2 Intel 公司 X86 体系结构嵌入式芯片简介

在当今世界微处理器市场上，Intel 公司的 X86 系列芯片产品在整个微处理器市场上获得约 80%的份额，具有压倒一切的优势。同时，Intel 公司也推出了基于 X86 系列产品的嵌入式处理器芯片。Intel 公司嵌入式芯片主要有以下几个系列：i960 系列、嵌入式 386/486 系列，以及最新的基于 StrongARM 核的 SA110、SA1100、SA1110 系列和 2002 年才推出的 Xscale 系列等。

i960 系列的处理器是 32 位的嵌入式超标量体系结构 RISC 处理器。i960 系列处理器能提供高性能的 32 位解决方案。它的产品系列从低端的 i960 S 系列到运算速率达 166MIPS 的 i960 H 系列。i960 系列中的 i960CA 是世界上第一个 32 位的嵌入式超标量体系结构处理器，它采用多发射技术，每时钟周期发射两条指令，使得 RISC 结构技术的性能加倍，i960CA 能达到 66MIPS 的性能。

嵌入式 386/486 系列产品，则是对应的桌面产品的嵌入式版本。Intel 对 386/486 系列产品进行了加强，降低功耗，提高抗干扰能力，放宽其工作条件限制，使其满足嵌入式应用的需要。该系列产品均为 32 位内核，与 PC 兼容，这使得该系列产品可以方便地重用运行在传统 PC 机上的代码。为嵌入式系统的设计带来了极大的方便。80386 内核提供了许多新的特性，包括存储器保护和多任务支持，并且在保护模式下，80386 可以访问 64TB 的虚拟内存空间，这些增强的功能为实现复杂、高可靠的实时软件提供了支持。嵌入式 386/486 系列产品为嵌入式应用提供了有效彻底的解决方案，降低设计的复杂性，减少软件的开发

时间。其典型产品包括 80386SX、80386DX、80386SXSA、80386CXSA、80386CXSB、80386EXSA、80376、Intel486 SX、Intel486 GX 处理器等。

特别值得一提的是 StrongARM 系列产品，它原来是 DEC 公司和 ARM 公司合作开发的。1998 年，Intel 收购了 DEC 的处理器设计部门，从而获得了 StrongARM 的授权。此后，在 StrongARM 的基础上，Intel 又开发出 SA-110、SA-1100 和 SA-1110 等一系列 StrongARM 高性能嵌入式处理器。特别是 SA-1110，其功能极其强大，已经把液晶控制器、外设接口（USB、IrDA、UART、PCMCIA）、音频编解码器 Cdec 等和 CPU 集成到同一个芯片内，从而可以很方便地嵌入于各种掌上设备。Intel 最新的 Xscale 核心是 StrongARM 的改进产品，是下一代 StrongARM 芯片的发展基础。Xscale 与 StrongARM 相比，可大幅降低工作电压并且获得更高的性能。具体来讲，在目前的 StrongARM 中，在 1.5V 下可以获得 166MHz 的工作频率，在 2.0V 下可以获得 233MHz 的工作频率；而采用 Xscale 后，在 0.75V 时工作频率达到 150MHz，在 1.0V 时工作频率可以达到 400MHz，在 1.65V 下工作频率则可高达 800MHz。当在 0.75V 下以 50MHz 的工作频率运行时，其功耗相当于用一只 5 号电池连续工作一个星期。可见 Xscale 结构在能耗方面具有巨大优势。基于 XScale 技术开发的微处理器，可用于手机、便携式终端（PDA）、网络存储设备、骨干网（Backbone）路由器等设备。

1.4.3 ARM 公司嵌入式芯片简介

成立于 20 世纪 80 年代的英国专业处理器设计公司 ARM 公司，是目前最成功的处理器 IP Core 提供商。ARM 公司旗下没有一间生产工厂，也不自己销售芯片。它的盈利方式是专注于高性能、低价格、低功耗处理器的设计，然后通过转让和授权生产 ARM 微处理器而获利。相对于同时代的其他嵌入式处理器，ARM 处理器能兼顾到高性能、低功耗、低价格等众多优势。

ARM 系列的处理器的最早研发开始于 20 世纪 80 年代中期，ARM6 的推出，将 ARM 引向成功。ARM7 是 ARM 授权最广泛的一个产品。在此之后，ARM 公司相继推出了 ARM8、ARM9、ARM10 等产品，而与 DEC 公司联合开发并授权 Intel 公司制造生产的 StrongARM 则是一种具有很高性能的通用微处理器。Intel 公司用 SAXXXX 命名该系列处理器。而且，Intel 最近推出的新一代 Xscale 处理器也是基于 StrongARM 的。

ARM7 是一种小型的高性能、低功耗、可集成 32 位 RISC 处理器内核。最初是为便携式通信设备而开发的，其经典产品 ARM7TDMI 是目前 ARM 结构中一种授权最广泛的产品，它把 ARM7 的指令系统与 Thumb 16 位精简指令集相结合，减少了存储器的容量，降低了系统成本。Thumb 指令集有着卓越的代码效率，意味着同等功能的执行代码，对存储器容量需求降低，使得利用 16 位宽度的存储器就可以达到 32 位存储器才有的高性能。

ARM9TDMI 是一种有 5 级流水线，集成有 Thumb 16 位精简指令集扩展功能、调试功能和哈佛结构总线的内核。在同样的工艺条件下，它的性能是 ARM7TDMI 性能的两倍以上。其典型应用为网络通信和机顶盒市场。

ARM10TDMI 是一种比 ARM9TDMI 有着更高性能的内核，其性能几乎是 ARM9TDMI



的两倍，有 6 级流水线、哈佛结构总线、Thumb 16 位精简指令集扩展和对所有编程模型状态的全调试访问。主要应用于下一代手持通信产品和数码电子消费产品以及多媒体应用。

综上所述，可以看出，ARM 系列处理器内核开发速度之快，性能之优越，是同时期的其他处理器产品无法比拟的。并且，上述的这些产品都提供了相应的软 IP 核（soft IP core），有特殊应用要求的 ASIC 厂商，可以购买这些软 IP 核，用于自己的产品开发。目前，中兴通信公司已经购买了 ARM 的软核，用于自己通信芯片的开发。使用软 IP 核可以减少开发的成本和风险，并加快产品的上市时间。

1.5 小结

本章是嵌入式 Linux 开发的入门篇章，主要介绍了有关嵌入式 Linux 开发的入门知识。通过本章的学习，读者应当对嵌入式系统的应用场合、嵌入式系统的定义和特点、如何选择适合的嵌入式操作平台、实时和实时系统的要求以及嵌入式 Linux 的特点有一定的了解。有过在其他嵌入式操作平台上开发经验的用户请仔细比较商用嵌入式操作系统与嵌入式 Linux 操作系统两者的异同。

同时由于嵌入式开发工作往往是与硬件紧密结合的，要求开发者对硬件平台有一定的了解，在本章中同时也对一些应用广泛的嵌入式芯片作了相应的介绍，如 Motorola、Intel 和 ARM 公司的产品。

1.6 思考题

1. 什么是嵌入式系统？它的特点是什么？
2. 请举出身边的嵌入式系统的例子。
3. 什么是实时系统？评价它的指标是什么？
4. 请解释中断禁止时间和中断延迟时间的区别和联系。
5. 和通用 RTOS 相比，嵌入式 Linux 的特点是什么？
6. 你曾在哪种硬件平台上开发过产品？它的特点是什么？

第2章　开发嵌入式 Linux 应用软件

知识点：

- 嵌入式 Linux 开发平台
- 开发嵌入式 Linux 软件所需的编辑、编译和调试工具
- 了解嵌入式 Linux 软件的开发流程

本章导读：

本章将主要介绍嵌入式 Linux 应用软件开发的基础知识，包括建立嵌入式 Linux 开发平台，使用 Linux 开发工具编辑、编译和调试软件。同时，对不具有嵌入式软件开发经验的读者来说，了解嵌入式软件的开发流程也是必须的，故本章也作了一些介绍。为了使读者更好地掌握以上内容，本章将以 uClinux 为平台，通过一些实例来说明。



2.1 建立嵌入式开发平台

在开发嵌入式应用软件之前，应当建立自己的开发平台，以后所有的开发工作都是在这个平台下进行的。

2.1.1 嵌入式开发平台简介

由于嵌入式系统自身的特殊性，注定了它自身所具有的资源和内存空间都是十分有限，不可能像开发 PC 软件那样在其上运行所有的开发工具，而且很多嵌入式系统都没有像显示器那样的输出设备，这些都决定了嵌入式软件开发应当采用一种特殊的模式：主机-目标机模式，使用交叉开发的方式进行开发。

其中主机就是常用的 PC 或工作站，嵌入式实时开发环境就运行在主机上，主机的操作系统是通用的 Windows 或 Linux 系统。目标机就是嵌入式应用系统，它运行嵌入式实时操作系统，和主机通过串行口、以太网、仿真器或其他方式通信，下载运行在主机中编译好的代码。主机-目标机的开发模型如图 2-1 所示。



图 2-1 主机-目标机开发模型

开发环境运行在主机上，用户所有的开发工作都是在主机开发环境中进行，包括编辑源代码、编译、连接、下载和调试等。生成的可执行目标代码通过串行口或以太网口下载到目标机，在目标机执行时，可以把执行结果回显到主机上，主机还可以通过开发环境提供的调试工具对代码进行调试。

RTOS 是嵌入式应用软件的运行基础，它与硬件平台息息相关。在单片机嵌入式系统中是没有操作系统的，但它仍需要有一个主程序对各任务进行调度分配。随着系统越来越复杂，需要管理的资源越来越多，没有操作系统对之进行调度管理是不行的。操作系统向应用软件提供访问资源的 BSP（板级支持包），这样就可以屏蔽部分硬件，使应用软件开发者不必关注这些硬件细节，减小开发难度。同时，通过 RTOS，还可以提高应用软件在不同硬件平台间的代码重用率。

在嵌入式应用软件开发中，目前大部分使用 C 语言作为开发工具，C 语言作为一种高效通用的高级语言，大大提高了开发效率，缩短了产品开发时间。与一般的 C 语言编译器

不同的是，嵌入式 C 语言编译器一般都经过优化，以提高编译效率。另外由于嵌入式处理器速度的提高和存储器空间的增长，一些嵌入式平台也把面向对象的语言 C++作为开发工具。在手机和 PDA 应用软件的开发中，J2ME 的出现也是一大亮点。

在线仿真器（ICE）是进行嵌入式系统调试最有效的开发工具。它首先通过实际执行，对应用程序进行原理性检验，以排除程序中存在的逻辑错误。在线仿真器的另一大功能是在应用系统中仿真微控制器的实时执行，发现和排除由于硬件干扰等引起的异常行为。此外，高级的 ICE 带有完善的跟踪功能，可以实时跟踪嵌入式系统的当前状态、微控制器对状态变化的反应，以及应用系统对控制命令的响应等一系列动作，这对于开发调试是非常重要的。

在嵌入式系统开发环境中，交叉调试工具是必需的，交叉调试工具用于在主机上调试目标机上运行的程序。在目标机上运行一个代理，以接收主机发送过来的命令和代码，并且解释执行。通过调试器，开发者可以设定程序运行的起止位置和断点，同时可以查看和改变变量、寄存器和内存中的值，设置程序运行条件等。使用户感到如同在本机上调试程序一样方便。

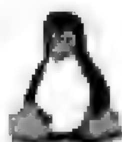
2.1.2 uClinux 简介

uClinux 是当前广泛应用的一种嵌入式 Linux 操作系统，也是本书选定的实例开发平台，uClinux 是一个完全符合 GNU/GPL 公约的项目，与 UNIX 系统兼容，完全开放源代码。英文单词中 u 表示 Micro，“小”的意思，C 表示 Control，“控制”的意思，所以 uClinux 字面上的理解就是“微控制领域中的 Linux 系统”。uClinux 现由 Lineo 公司支持维护，官方主页为 www.uclinux.org。

uClinux 针对嵌入式应用的特点，对 Linux 2.0 内核进行了修改和重新编译，其大小远小于原来。它包含 Linux 常用 API，但是内核小于 512KB，并且保留了原来 Linux 操作系统所具有的高稳定性、强大的网络功能和卓越的文件系统支持功能等优点。

uClinux 的一大特点是它没有 MMU（Memmory Management Unit，内存管理单元），它是专门针对没有 MMU 的 CPU 而设计的，并专为嵌入式系统做了许多小型化的工作。目前已支持 CPU 的芯片包括 Motorola 公司的 68k 系列、PowerPC 系列以及 ARM 公司的一系列芯片等。

uClinux 是专门针对没有 MMU 的处理器而设计的，它不能使用处理器的虚拟内存管理技术，但出于移植简单和尽量靠拢标准 Linux 的需要，uClinux 仍然沿用标准 Linux 的分页内存管理结构，系统在启动时将实际存储器进行分页，但实际上采用的是实存储器管理策略。uClinux 系统对于内存的访问是直接的，它对地址的访问不需要经过 MMU，而是直接送到地址线上输出，所有程序中访问的地址都是实际的物理地址。操作系统对内存空间没有保护（这实际上是很多嵌入式系统的特点），各个进程实际上共享一个运行空间，没有独立的地址转换表。一个进程在执行前，系统必须为进程分配足够的连续地址空间，然后全部载入主存储器的连续空间中。



由于没有 MMU 提供对内存空间的保护，对于 uClinux 应用系统的开发人员来说，他们必须参与系统的内存管理，开发人员必须明白自己程序的运行位置及所需空间的大小，并且保证不会破坏其他程序的运行空间以及系统的稳定。这样可能会增大开发难度，但对于嵌入式系统来说是很正常的，因为这样可以提高系统效率。因为如果运行 MMU，势必要占用大量的系统资源。

uClinux 支持多种文件系统，包括 NFS（网络文件系统）、ext2（Linux 文件中事实上的标准）、ROMFS、JFFS、MS-DOS 及 FAT16/32 等。通常采用的文件系统是 ROMFS，这种文件系统相对于一般的 ext2 文件系统所需要的空间更少。ROMFS 文件系统不支持动态擦写保存，对于系统需要动态保存的数据采用虚拟 RAM 盘/JFFS 的方法进行处理（RAM 盘将采用 ext2 文件系统）。

uClinux 小型化的另一个做法是重写应用程序库，相对于越来越大且越来越全的 glibc 库，uClibc 对 glibc 作了精简。uClibc 是一个用于开发嵌入式 Linux 系统的 C 程序库，它大大小于 GNU C 程序库（glibc），但是几乎所有 glibc 支持的应用 uClibc 也可以完美支持。uClibc 由 Erik Andersen 维护，官方网址为：http://uclibc.org/uClibc.html。对于程序库的使用，通常采用静态连接的方法，这是由 uClinux 内存管理方式所决定的。由于技术人员的努力，目前 uClinux 已经支持动态连接和 ELF。

对于 uClinux 的实时性问题，uClinux 自身不提供实时性，但是它可以采用 RT-Linux 的实时解决方案。RT-Linux 任务调度管理器把普通 Linux 的内核当成一个任务运行，同时调度管理其他实时任务，而对于所有的非实时进程则交给普通 Linux 内核处理。这种方式对于解决普通操作系统的实时性问题是个很好的解决方案，实现简单，且实时性容易检验，与原来的版本保持了兼容性。uClinux 可以使用 RT-Linux 的实时补丁，以增强实时性，使其可以应用于工业控制等一些实时性要求较高的场合。

2.1.3 uCsimm

对于 uClinux 的硬件平台，下面介绍比较典型的微控制模块 uCsimm。

uCsimm 是专门针对 uClinux 操作系统而构造的微控制模块，它采用 Motorola 公司 DragonBall 系列 68EZ328 微处理器。标准 uCsimm 自带 2MB 的 FLASH 和 8MB 的 DRAM。另外还带有一个 10Base-T 以太网口和一个 RS-232 串行口，并带有内置的 LCD 驱动，能够在 QVGA 的方式下显示 320×240 像素。uCsimm 的外观如图 2-2 所示。



图 2-2 uCsimm 外观

uCsimm 的外观参数如表 2-1 所示。

表 2-1　uCsimm 外观参数

外　观	物 理 参 数
长	25 英寸
宽	3.5 英寸
高	1 英寸
管脚	标准 30 针

uCsimm 的体积非常小，带有 30 针输出。由于嵌入式系统需要良好的稳定性和以太网功能，对空间要求严格，而且所处理的任务比较单一明确，所以如此小体积的 uCsimm 特别适合嵌入式应用，并且已经在网络服务器和可编程逻辑器中得到广泛应用。

uCsimm 用到的主要元器件为：

- CPU：Microprocessor MC68Z328 DragonBall。
- RAM Memory：8 MB DRAM（4096K×16 bit）。
- FLASH ROM：（可选）。
- 1MB（512K×16 bit）。
- 2MB（1024K×16 bit）。
- 4MB（2048K×16 bit）。
- 以太网口：10baseT。

2.1.4　建立 uClinux 开发平台

下面将详细介绍 uClinux 开发平台的建立，作为例子，将选用在国内嵌入式 Linux 领域处于领先地位的华恒公司的产品——HHPPC860-3COM-2ETH-R1。

华恒 HHPPC860-3COM-2ETH-R1 套件是一套完整的基于摩托罗拉 MPC860 处理器的嵌入式开发平台。MPC860 Power QUICC（Quad Integrated Communications Controller）内部集成了微处理器和一些控制领域的常用外围组件，特别适用于通信产品。它内部集成了两个处理模块：一个是嵌入的 PowerPC 核，另一个是通信处理模块（CPM，Communications Processor Module），由于 CPM 分担了嵌入式 PowerPC 核的外围工作任务，这种双处理器体系结构功耗要低于传统的体系结构的处理器。

华恒 HHPPC860-3COM-2ETH-R1 套件由核心板和底板组成，在核心板上集成了摩托罗拉 MPC860 处理器、16MB SDRAM 以及 4MB 的 FLASH，为用户的软件研发提供了足够的空间。在底板上则提供非常丰富的外设接口：一个 10Mbit/s 以太网接口、两个两线 RS-232 串口（COM1、COM2）、一个四线 MODEM 拨号 RS-232 串口（COM3）、100 Mbit/s 快速以太网口及一个 BDM 调试口。核心板和底板配合即构成一个最小的完整应用系统。该系统具有体积小、耗电低、处理能力强、网络功能强大等特点，能够装载和运行嵌入式 Linux 操作系统。用户可以在这个系统平台上进行自主软件开发，并对 MPC860 进行测试和评估，也可在保持



核心板不变的情况下，针对具体的应用通过对底板的更改来定制自己的应用系统。

1. 华恒 HHPPC860-3COM-2ETH-R1 套件的安装

下面将介绍华恒 HHPPC860-3COM-2ETH-R1 套件的安装步骤：

（1）在一台 PC 上安装 Linux。

推荐使用 RedHat 7，选择 Custom 定制安装，在选择软件 Package 时选择最后一项：everything，即完全安装。

（2）配置好网络、TFTP 服务和 NFS（Enable Running）。

网络配置主要是要安装好以太网卡，对于一般常见的 RTL8139 网卡，RedHat 7.2 可以自动识别并自动安装好，完全不要用户参与，因此建议使用该网卡。然后配置宿主机 IP：ifconfig eth0 192.168.1.199。其中 TFTP 服务对于以后步骤用以太网下载烧写 FLASH 是必需的，而 mount 命令也需要配置好 NFS。另外，将用到 Linux 自带的串口通信程序 minicom，通过它可以监视并控制目标板，在此把串口通信速率设为 115200bit/s。

（3）将厂商附带的光盘插入光驱，用 mount 命令加载光驱，然后使用 cce（或其他中文环境，如 gce）进入中文环境（cce 可以在 sourceforge 网站下载，它的 URL 是 http://sourceforge.net/projects/cce2k）。再执行光盘安装程序。

注意：进入中文环境只是为了能够看到安装启动时的一些中文提示信息。若没有中文环境也无所谓，只是看到一些乱码而已，用户只需按一下“y”，按回车键即可完成全部安装。

（4）使用下列命令安装 uClinux。

./ppcinst

执行完毕后，会在根目录下生成工作目录：/LinuxPPC。以下对这个工作目录下的各子目录功能进行说明：

/LinuxPPC/usr/src/linux/　Linux 内核源代码。

/LinuxPPC/usr/src/application/　应用程序源代码。

/LinuxPPC/usr/src/tarballs/　工具、编译器、库函数和 uClinux 内核源代码。

/LinuxPPC/usr/src/ppcboot-1.15　RAM 版 PPCboot 的映像文件和 ROM 版 PPCboot 的映像文件及源代码（华恒公司不提供 RAM 版 PPCboot 的源代码）。

/LinuxPPC/CDK/bin/　Linux 内核编译及转换工具。

/LinuxPPC/CDK/bdmtools/　BDM 工具及 gdb 源代码。

（5）使用 BDM 工具下载 RAM 版的 PPCboot 烧写工具。

BDM 工具完成板卡硬件检测、下载、运行、烧写 FLASH、内核调试、单步调试等底层的调测功能。PPCboot 用于完成目标板启动时的初始化，它可以分为 RAM 版和 ROM 版。RAM 版的 PPCboot 不是烧制在目标板的 FLASH 中，而是每次启动时在 BDM 命令模式下从主机加载到目标板的 RAM 中运行。

其工作过程为：检查连接是否正确，接通 BDM 和目标板电源后进入 BDM 提示命令行状态。第一次上电初始化时在 BDM 提示命令行状态下输入如下命令：

setupproc，回车

q，回车

y，回车（确定退出，返回 Linux 命令提示符状态）

继续在 Linux 命令提示符下输入：

./powerpc-linux-gdb -x MPCBDM/mpc.init

在 BDM 提示命令行状态下输入如下命令：

doinit

以上命令输入后会出现初始化信息，并加载 RAM 版 PPCboot 到 RAM 的 0xF00000 处，运行 RAM 版 PPCboot：

j *0xf00100

从 0xF00100 处运行 RAM 版的 PPCboot。

powerpc-linux-gdb 调用 mpc.init 脚本对 CPU 进行初始化（sys-init），然后下载 RAM 版 PPCboot 到目标板的 RAM 中，并让它运行起来，这时 CPU 完全由 PPCboot 接管。PPCboot 运行后初始化以太网及 TFTP 协议栈。

这时可以切换到 minicom 的工作界面，在宿主机的 minicom 终端窗口就可看到打印出的启动 PPCboot 的全部信息和命令提示符。RAM 版本的 PPCboot 提供了一些测试工具，如查看存储器、寄存器信息命令和检测 SDRAM 等。具体使用方法可以使用 help 命令进行查看。

2. 烧写 ROM 版 PPCboot。

如果已经烧写过 ROM 版的 PPCboot（目标板出厂时已烧写），则可以不用再烧写，跳过这一步直接烧写用户升级修改过的 Linux 内核。

烧写 ROM 版 PPCboot 过程如下：

首先，启动目标板进入 BDM 模式并加载运行 RAM 版 PPCboot。在宿主机一侧将 ROM 版 PPCboot 的映像文件 ppcboot.bin 复制到 /tftpboot 目录下或在当前目录下（/LinuxPPC/usr/src/ppcboot-1.1.5/）make 重新生成 ROM 版 PPCboot 的映像文件 ppcboot.bin，并复制到/tftpboot 目录下：

cp /LinuxPPC/usr/src/ppcboot-1.1.5/ppcboot.bin /tftpboot/

然后，在 minicom 中的 PPCboot 命令提示符下输入 tftp 0 ppcboot.bin。将 ROM 版 PPCboot 通过 TFTP 服务器下载到 SDRAM 的起始地址 0x0 处，并烧写到 FLASH 中。

整个烧写过程在 minicom 中将打印如下信息：

```
=> tftp 0 ppcboot.bin
ARP broadcast 1
Got good ARP - start TFTP
TFTP from server 192.168.1.199; our IP address is 192.168.1.122
Filename 'ppcboot.bin'
Load address: 0x0
Loading: ################################
done
Program FLASH... ...
```




```
Erase FLASH... ...
Bytes transferred = 143724 (2316c hex)
bd_ptr address is = 0xf9ffc4
File size :143724 (2316c hex)bytes
######
Write FLASH suc !
bd_ptr address is = 0xf9ffc4
bd_ptr address is = 0xf9ffc4
bd_ptr address is = 0xf9ffc4
bd_ptr address is = 0xf9ffc4
```

注意：一定要看到显示"Write FLASH suc !"才表示烧写完成了。如果 # 长时间停止，则表示 FLASH 可能有问题，建议检查 FLASH。

3. 烧制嵌入式 Linux 系统

烧制过程如下：进入/LinuxPPC/usr/src/linux 目录下，运行./easy_build，编译 Linux 内核映像文件 linux.bin，生成 Linux 内核映像文件 linux.bin 并自动复制到/tftpboot 目录下，可以查看文件信息，确定为最新生成的 Linux 内核映像文件。

进入 BDM 模式并加载运行 RAM 版 PPCboot，在 minicom 中的 PPCboot 命令提示符下输入 tftp 0 linux.bin，将 Linux 内核映像文件 linux.bin 通过 TFTP 服务器下载到 SDRAM 的起始地址 0x0 处，并烧写到 FLASH 中。整个烧写过程在 minicom 中将打印如下信息：

```
=> tftp 0 linux.bin
ARP broadcast 1
Got good ARP - start TFTP
TFTP from server 192.168.1.199; our IP address is 192.168.1.122
Filename 'linux.bin'.
Load address: 0x0
Loading:
#################################################################
done
Program FLASH... ...
Erase FLASH... ...
Bytes transferred = 2243532 (223bcc hex)
bd_ptr address is = 0xf9ffc4
File size :2243532 (223bcc hex)bytes
#################################################################
#########
Program FLASH !Please wait ... ...
```

```
Program FLASH done !
bd_ptr address is = 0xf9ffc4
bd_ptr address is = 0xf9ffc4
bd_ptr address is = 0xf9ffc4
bd_ptr address is = 0xf9ffc4
bd_ptr address is = 0xf9ffc4
bd_ptr address is = 0xf9ffc4
```

看到"Program FLASH done !"表示烧制成功。

目标板操作系统加载成功后，当启动 RAM 时的内在分配情况如图 2-3 所示。

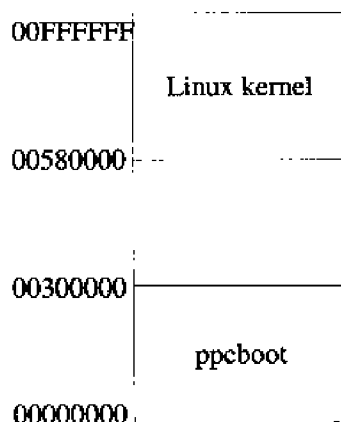


图 2-3　内存分配图示

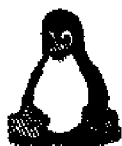
至于如何获取最新版本的 uClinux 源代码，编译建立 uClinux 内核映像文件 linux.bin。获取 uClinux 源代码的方式有两种：直接从互联网上下载或定购正式发行的 CD-ROM（在华恒的开发套件中已附赠 uClinux 源代码，这步可以省略）。

若是从互联网上下载，可以从 http://www.uclinux.org/pub/uclinux 处下载，各版本的源代码及补丁都有。

通过以上的步骤后，就建立了一个嵌入式 Linux 开发平台。至此，即可在这个平台上开发自己的嵌入式应用了。

如果不使用交叉调试，嵌入式 Linux 应用程序的开发有两种模式：

- 先在宿主机（Intel CPU）上调试通过后，再移植到目标板（MPC860）上。但这种开发模式有两个问题：首先，宿主机的运行环境可能与目标机不同，如 CPU 的体系结构就可能不一样，宿主机不能完全模拟目标机，对于一些如外部设备中断服务程序之类的程序也没办法调试；其次是函数库的问题，在程序移植时可能会存在函数未定义的问题。对于这种问题，一般要求开发者自己编制这些要用到却又未定义的函数。
- 直接在目标板上进行开发（通用开发模式，建议采用该模式）。在这种模式下将宿主机和目标板通过串口或网口相连，在宿主 PC 机上运行 minicom 作为目标板的显示终端，mount 宿主机硬盘到目标机中，直接在目标板上调试应用。例如：



mount -o nolock 192.168.1.199:/ /mnt （mount 开发宿主机的根目录到目标板的/mnt 目录下）；

cd /mnt（/mnt 目录下即为开发宿主机硬盘根目录下的内容）；

./hello（执行 hello 程序，hello 为开发宿主机硬盘根目录下编译后的可执行文件，具体编译调试可执行文件的方法将在本章 2.2 节介绍）。

2.2 嵌入式 Linux 软件开发工具

下面将介绍开发嵌入式 Linux 应用软件所用到的一些常用工具，包括编辑工具 vi、编译工具 gcc、MakeFile 文件及调试工具 gdb。

2.2.1 使用 vi 编辑器

vi 是 Linux/UNIX 世界里极为普遍的全屏幕文本编辑器，几乎任何一台 Linux/UNIX 机器都会提供这个软件。这种编辑器的一大好处是所有的命令按键都在手指范围内，手不必离开主键盘就可输入所有命令。

vi 有 3 种状态，即可视命令模式、冒号命令模式及文本编辑模式。

一般当进入 vi 时，会首先进入可视命令方式，这是 vi 的启动默认模式。在这种模式下用户输入的任何内容都被认为是编辑命令。如按下"i"键就进入插入方式，可从光标左侧输入文本；如按下"a"键就进入添加方式，可以在光标右侧输入文本。

在冒号命令方式下，所有命令都要以":"开始，所输入的字符系统均作命令来处理，如输入":q"代表退出，":w"表示存盘。

进入文本编辑模式，这时用户可以进行所有文本的编辑操作。在文本编辑模式下，按下 Esc 键就可以回到命令状态。

无论是创建新文件或修改旧文件，都可以使用 vi，所需指令为：

vi filemane

如果文件是新的，就会在屏幕底部看到一个信息，告诉用户正在创建新文件。如输入命令：vi /tmp/test。若是新文件，就应该看到下列类似信息：

```
~
~
~
~
~
"/tmp/test" [New File]
```

以上是一个经 vi 创建的事例文件，一行开始处的波折号"～"表示文件这一行是空行。如果文件早已存在，vi 则会显示文件的前 24 行中的内容。这时用户就可以使用下列命令进

入编辑模式：

- 指令按键“i”：在光标处前面插入正文，光标后的文字随追加的文字向后移动。
- 指令按键“I”：在光标所在行开始处插入正文。
- 指令按键“a”：在光标所在位置后面追加正文，光标后的文字随追加的文字向后移动。
- 指令按键“A”：在光标所在行行尾追加正文。
- 指令按键“o”：在光标所在行下面新开一行，并进入编辑状态。
- 指令按键“O”：在光标上面新开一行，并进入编辑状态。

下面对一些基本命令加以解释：

（1）光标命令。

- k、j、h、l：上、下、左、右光标移动命令。虽然也可以在 Linux 中使用键盘右边的 4 个光标键，但是这 4 个命令还是非常有用的，因为这 4 个键正是右手在键盘上放置的基本位置。
- nG：跳转命令。n 为行数，该命令立即使光标跳到指定行。
- Ctrl+g：光标所在位置的行数和列数报告。
- w、b：使光标向前或向后跳过一个单词。
- 0：移动光标到所在行的最前面，相当于功能键 Home。
- $：移动光标到所在行的最后面，相当于功能键 End。
- Ctrl+d：光标向下移半页。
- Ctrl+f：光标向下移一页。
- Ctrl+u：光标向上移半页。
- Ctrl+b：光标向上移一页。
- H：移动到屏幕的第一行。
- M：移动到屏幕的中间行。
- L：移动到屏幕的最后行。

（2）编辑命令。

- i、I、a、A：在说明如何进入编辑模式时已经介绍。
- r、R：r 修改光标所在字符，r 后接要修改的字符。R 进入取代状态，新增资料会覆盖原先资料，直到按 Esc 键回到指令模式下为止。
- cw、dw：改变（置换）/删除光标所在处的单词的命令（c=change、d=delete）。
- x、X：删除光标所在处后面/前面的字符。
- d$、d0：删除光标所在处到行尾/行首的命令。
- dd：删除光标所在行。
- dw：删除光标处的单词。
- nx：删除光标处后 n 个字符。
- nX：删除光标处前 n 个字符。
- ndw：删除光标处后 n 个单词（word）。
- u：恢复前一次所做的操作，相当于 Word 工具中的 undo 操作。



- y：复制操作（yank）。
- p：粘贴操作（普通）。

（3）查找/替换命令。

- /string、?string：从光标所在处向前或向后查找相应的字符串的命令。
- n：同一方向下重复检索。
- N：相反方向上重复检索。
- rchar：由 char 代替光标处的字符。
- Rtext：由 text 代替光标处的字符。
- Cwtext：由 text 取代光标处的单词。
- Ctext：由 text 取代光标处至该行结尾处。
- cc：使整行空白，但保留光标位置，开始输入。
- :%s/string1/string2/g：在整个文件中替换“string1”成“string2”。

下面通过实例说明查找/替换命令的使用：

/hello：向前查找 hello 字符。

? goodbye：向后查找 goodbye 一词。

/The* foot：向前查找 The 一词所在的行以及 The 之后的某一点上的 foot 词汇。

? [pP]rint：向后查找 print 或 Print 单词（注意在 Linux 中大小写是严格区分的）。

:%s/Local/s/Remote/g：用 Remote 一词替换文件中每一行的 Local。

（4）存盘/退出命令。

在修改文件时，如何存档及退出指定文件都非常重要。在 vi 内，使用存档或退出的指令时，要先按冒号（:），进入冒号命令模式，用户就可以看见在屏幕左下方，出现冒号“:”，这表示 vi 已经进入冒号命令模式，在此可以完成存档或退出等工作。以下是这种模式下的一些常用命令。

- :q!：放弃任何改动而退出 vi，也就是强行退出。
- :w：存档不退出。
- :w!：对于只读文件强行存档。
- :wq：存档并退出 vi。
- :x：与 wq 的工作一样。

注意：*在编辑模式下，不能输入指令，必需先按下 Esc 键，返回命令模式。假若用户不知身处何态，也可以按下 Esc 键，这时不管处于何态，都会返回命令模式。*

vi 还可以同时编辑多个文件，它的使用方法是：vi filelist。如同时编辑 2 个文件，复制一个文件中的文本并粘贴到另一个文件中，命令如下：

vi file1 file2，回车

yy，回车（在文件 1 的光标处复制所在行）

:n，回车（切换到文件 2（n=next）或者按 Ctrl+ww，就在两个文件间切换）

p，回车（在文件 2 的光标所在处粘贴所复制的行）

:n，回车（切换回文件 1）

如果要在 vi 执行期间，转到 shell 执行，使用惊叹号（!）将执行系统指令。例如在 vi 的命令状态，列出当前目录内容，可以输入：

:!ls

另一方面，用户可以在主目录中创建 exrc 环境文件，用 set 设置选项，每次调用 vi 时，都会读入.exrc 中的指令与设置。以下是 exrc 环境文件的应用实例：

```
set   wrapmarging=8
set   showmode
set   autoindent
```

2.2.2 使用 gcc 编译嵌入式 C 应用程序

运行于 Linux 操作系统下的自由软件 GNUgcc 编译器，不仅可以编译 Linux 操作系统下运行的应用程序和 Linux 本身，还可以作交叉编译，编译运行于其他 CPU 上的程序。可以作交叉编译的 CPU（或 DSP）涵盖了几乎所有知名厂商的产品。用于嵌入式应用的、众所周知的 CPU 包括：Intel 的 i386、Intel960、AMD29K、ARM、M32、MIPS、M68K、ColdFire、PowerPC、68HC11/12、TI 的 TMS32 等。详细列表可到 http://gcc.gnu.org/reading 网站查看。

GNU gcc 编译器是一套完整的交叉 C 编译器，包括：

- C 交叉编译器 gcc。
- 交叉汇编工具 as。
- 反汇编工具 objdump。
- 连接工具 ld。
- 调试工具 gbd。

可以用批处理文件 MakeFile 将上述工具组合成方便的命令行形式。

使用 gcc 编译 C 程序生成可执行文件有时似乎是一步完成的，但它其实要经历如下 4 个步骤：

（1）预处理（Preprocessing）。这一步需要分析各种命令，如#define、#include、#ifdef 等。gcc 调用 cpp 程序来进行预处理。

（2）编译（Compilation）。这一步将根据输入文件产生汇编语言，由于通常是立即调用汇编程序，所以其输出一般不保存在文件中，gcc 调用 ccl 进行编译工作。

（3）汇编（Assembly）。这一步中将汇编语言用作输入，产生具有.o 扩展名的目标文件。gcc 调用 as 进行汇编工作。

（4）连接（Linking）。在这一步，各目标文件.o 被放在可执行文件的适当位置上，该程序引用的函数也放到可执行文件中。gcc 调用连接程序 ld 来完成最终的任务。

gcc 命令的基本用法为：gcc [option] [filename]。命令行选项指定的操作将在命令行上每个给出的文件上执行。例如：

gcc -o prog main.c subr1.c subr2.c subr3.c。

其中，“-o prog” 指定输出的可执行文件名为 prog，如果没有指定-o 参数，gcc 将使用



默认的可执行文件名 a.out。如果想单独编译每一个源文件，最后再进行连接，可以调用如下命令：

```
gcc -c mian.c
gcc -c subr1.c
gcc -c subr2.c
gcc -c sunbr3.c
gcc -o prog main.o sunbr1.o sunbr2.o sunbr3.o
```

其中，-c 选项表示编译产生目标文件，但不连接，最后将所有目标文件连接在一起，构成可执行文件。由于最后一个命令的输入都是目标文件，不需要编译和连接，所以 gcc 就只调用连接工具 ld。

gcc 的命令选项有许多项，但经常使用的几个选项是：

- -c：只预处理、编译和汇编源程序，不进行连接。编译器对每一个源程序都将产生一个后缀名为.o 的目标文件。
- -o Exefile：确定输出文件为 Exefile，如果没有指定 Exefile，默认输出为可执行文件 a.out。
- -Dmacro 或-Dmacro=defn：其作用类似于源代码程序中的#define。例如：
  % gcc -c -DHAVE_GDBM -DHELPFILE=\"help\" cdict.c
  其中，第一个-D 选项定义宏 HAVE_GDBM，在程序中可以用#ifdef 去检测它是否被设置；第二个-D 选项将宏 HELP_FILE 定义为字符串“help”（由于反斜线的作用，引号实际上已经成为该宏定义的一部分），这对于控制程序打开哪个文件是很有用的。
- -O：对程序编译进行优化，编译程序试图减少被编译程序的长度和执行时间，但其编译速度比不进行优化慢，而且要求较多的内存。在编译嵌入式应用软件时，如果主机的性能较好，可以打开这个选项。
- -O2：允许比-O 更好地优化，编译速度也较之更慢一些，但结果程序的执行速度比较快。
- -g：告诉 gcc 产生能被 GNU 调试器使用的调试信息以便调试程序。gcc 提供了一个很多其他 C 编译器里没有的特性，即在 gcc 里能使-g 和-O（产生优化代码）联用。这点非常有用，因为能在与最终产品尽可能相近的情况下调试代码。在同时使用这两个选项时必须清楚所写的某些代码已经在优化时被 gcc 作了改动。关于调试 C 程序的更多信息请看 2.2.4 节对 gdb 调试工具的介绍。
- -pg：这个选项告诉 gcc 在程序里加入额外的代码，执行时产生 gprof 用的剖析信息以显示程序的耗时情况。
- -Idir：将 dir 目录加到搜寻头文件的目录列表中去，并优先于在 gcc 中默认的搜寻目录。在有多个-I 选项的情况下，按命令行中-I 选项的前后顺序搜索。dir 可以是相对路径，如-I../inc 等。

2.2.3 编写 MakeFile

UNIX/Linux 系统上的很多软件包都是使用 make 程序和 MakeFile 文件来实现自动编译的。使用 make 程序的目的是自动确定一个软件包哪些部分需要重新编译，并用特定的命令去编译它。准确地使用 make 可以大大减少编译程序所花费的时间，因为它可以消除不必要的重编译。

要正确使用 make，必须编写 MakeFile 文件。MakeFile 文件描述了软件包中各个文件之间的依赖关系，提供了更新每个文件的命令。通常，一个可执行文件的更新需要它所连接的目标文件的更新，而目标文件由编译源文件而更新。

如果一个适当的 MakeFile 文件存在，当改变某些源文件后，只要在 shell 下使用 make 命令就可以完成所有必需的重新编译。make 程序利用 MakeFile 文件中的数据和每个文件最近一次更改的时间来确定哪些文件需要更新。对于需要更新的文件，make 程序使用 MakeFile 中定义的命令来更新。

至于使用哪个 MakeFile 文件来更新，可以在 make 命令中用-f 选项来指定。如果不指定，make 程序将在当前目录下按下列顺序寻找如下文件：GUNMakeFile、MakeFile 和 MakeFile。最好是使用 MakeFile，因为它的第一个字母是大写的，通常被列在一个文件目录的所有文件列表的最前面，便于查找。

1. 编写规则

下面对组成 MakeFile 文件的一些规则加以说明。在这之前首先介绍目标（target）及相关性的概念。

- 目标就是 make 程序要完成的一项任务，目标通常是一个文件的文件名，也有例外。
- 相关性即一个目标的完成依赖于其他一些目标或文件。

MakeFile 文件中包含着一些目标，对于每一个目标，都提供了与这个目标具有相关性的其他目标或文件的名字，以及实现这个目标的一组命令。它的书写规则是：

目标[属性]
分隔符号[依赖文件][；命令列]
{<tab>命令列}

与 Linux 下面的命令格式相同，[ ]中的内容表示可选，{}中的内容表示可以出现多次。下面对几个条目的意义进行说明：

- 属性：表示该目标文件的属性。
- 分隔符：用来分割目标文件和依赖文件的符号，如冒号“:”等。
- 依赖文件：实现目标所需要的文件的列表。
- 命令列：重新生成目标的命令，可以有多条命令。除第一条命令外，以后的每一条命令都必须以制表键 Tab 隔开。

下面通过一个简单的 MakeFile 文件的应用实例加以说明：

#一个简单的 MakeFile 的例子




```
#以#开头的为注释行
exeProg: main.o    subfunc.o
gcc -o    exeProg main.o subfunc.o
main.o:    main.c     main.h
gcc -c -I.-o main.o main.c
subfunc.o: subfunc.c
gcc -c -o subfunc.o subfunc.c
clean:
rm -f *.o
```

上面的 MakeFile 文件共定义了 4 个目标：exeProg、main.o、subfunc.o 和 clean。每个目标都是从最左边开始写，后面跟一个冒号（:），如果这个目标的实现依赖于其他的目标或文件，把它们列在冒号的后面，并以空格隔开。然后另起一行开始写实现这个目标的一组 shell 命令。shell 命令可以有若干行。

注意：每个 shell 命令的第一个按键必须是字符或数字，不能以空格开头，否则 make 就会显示如下出错信息：MakeFilc:2:*missing seperator.Stop。**

一般情况下，调用 make 可以输入：

% make target

target 是 MakeFile 文件中定义的目标，如果省略 target，make 就将更新 MakeFile 文件的第一个目标。例如在上一个实例中，若输入 make 命令不带 target 参数，将更新目标 exeProg。

make 在检查一个目标是否已经过时并需要更新时，采用的是按相关性递归的方法。make 在构建一个目标之前要生成该目标所依赖的所有文件，并递归地前进，从而确保这些文件都是新的。具体构建目标的过程如下：

（1）如果一个目标 task 不是作为文件而存在，那它就已经过时了，命令 make task 必定执行该任务。

（2）make 检查所有与 task 有相关性的目标。对于不是 MakeFile 中定义的任务，而只是文件的相关目标，则检查相关目标的生成日期是否比 task 文件的生成日期更近，如果有一个更近则 task 就过时了。对于 MakeFile 定义为任务的相关目标，则按同样的方法检查其是否过时，如果任意一个过时了，都需要更新。

（3）递归从底层向上，对所有已经过时的目标进行更新，只有当一个目标所依赖的所有目标都已经更新后，这个目标才被更新。

通过上面的例子来说明目标的更新过程。在这里假设修改了文件 subfunc.c，可用以下命令更新目标 exeProg，即重新编译可执行文件 exeProg。

% make exeProg

由于 exeProg 依赖于目标 main.o 和 subfunc.o，所以必须检查 main.o 和 subfunc.o 是否已经过时。目标 main.o 依赖文件 main.c 和 main.h，比较目标文件 main.o 和源文件 main.c、main.h 的更新日期，如发现 main.o 比它所依赖的文件的日期更新，即不过时。再检查目标 subfunc.o，它依赖文件 subfunc.c，由于已经修改了 subfunc.c，它比 subfunc.o 更新，即 subfunc.o



过时了，从而依赖 subfunc.o 的所有目标都过时了。应该用在 makfile 文件中定义好的如下 shell 命令更新它：

gcc -c -o subfunc.o subfunc.c

由于目标 subfunc.o 过时并更新，导致目标 exeProg 已经过时，要完成“make exeProg”的任务，必须用定义 exeProg 的一组 shell 命令来更新它：

gcc -o exeProg main.o subfunc.o

如果是第一次编译上面这个软件，则因为 exeProg、main.o 和 subfunc.o 等目标文件都不存在，按照规定，这时所有的目标都是过时的，必须全部更新，而且必须从底而上执行定义这些目标的 shell 命令。

在上面的例子中，还定义了一个目标 clean，输入 make clean 命令将执行：

rm -f *.o

clean 的目标是 MakeFile 中常用的一种专用目标，即删除所有的目标模块。输入 make clean 命令时，make 程序将查看一个名为 clean 的文件，如果该文件不存在（约定永远不在文件目录中使用该名字作为文件名称），make 将执行定义该目标的所有命令。

2. 宏和隐含规则

为了简化命令的书写，在 MakeFile 中可以使用几个预先定好的缩写和定义一些宏（macro）。以下是几个经常用到的缩写：

- $@：代表该目标的全名。
- $*：代表已经删除了后缀名的目标名。
- $<：代表该目录的第一个相关目标名。

按照这样的缩写，2.2.3 中的例子可以改写为：

```
#一个使用缩写符的 MakeFile 例子
#以#开头的一行是注释行
exeProg: main.o   subfunc.o
            gcc -o $@ prog.o subfunc.o
main.o:  main.c   main.h
            gcc -c -I -o $@ $<
subfunc.o:  subfunc.c
            gcc  -c -o     $@ $*.c
clean:
            rm -f *.o
```

这类缩写对于编写默认的编译规则是很有用的。

GNU 的 make 工具除提供建立目标的基本功能之外，还有许多便于表达依赖关系以及建立目标的命令特色。其中之一就是变量或宏的定义能力。如果要以相同的编译选项同时编译十几个 C 源文件，而为每个目标的编译指定冗长的编译选项的话，将是非常乏味的。但利用简单的变量定义，可避免这种乏味的工作。

#为编译器定义一个宏名




```
CC = gcc
# 定义编译宏标志
CCFLAGS = -D_DEBUG -g -m486
# 创建一个目标文件
test.o: test.c test.h
$(CC) -c $(CCFLAGS) test.c
```

在上面的实例中，CC 和 CCFLAGS 都是 make 的变量。GNU make 通常称之为变量，而其他 UNIX 的 make 工具称之为宏，实际是同一个东西。在 MakeFile 中引用变量的值时，只需在变量名之前添加$符号，如上面的$(CC)和$(CCFLAGS)。

如果在 MakeFile 文件中没有给出从某一目标的相关文件构建这一目标的命令，GNU make 包含有一些内置的或隐含的规则，这些规则定义了如何从不同的依赖文件建立特定类型的目标。

例如，对下面的 MakeFile 文件中的内容：

```
#一个简单的使用默认规则的 MakeFile 例子
exeProg: main.o subfunc.o
          gcc -o exeProg main.o subr.o
clean:
          rm -f *.o
```

exeProg 的相关目标 main.o 和 subfunc.o 的构造规则如果没有定义，make 程序将使用隐含规则。默认的隐含规则中可以生成目标类型为.o 文件的相关文件类型有多种（如.c、.cc、.C、.p、.f 等)。make 程序将按顺序找出第一个存在的或可以构建的类型。例如，若它最先找到 main.c，它就使用隐含规则中的 main.c 构建 main.o，如果没有，就依此类推。

GNU make 支持两种类型的隐含规则，它们的表示方式为：

- 后缀规则（Suffix Rules）：后缀规则是定义隐含规则的较老的风格方法。后缀规则定义了将一个具有某个后缀的文件（例如.c 文件）转换为具有另外一种后缀的文件（例如.o 文件）的方法。每个后缀规则以两个成对出现的后缀名定义。例如，将.c 文件转换为.o 文件的后缀规则可定义为：

  ```
  .c.o:
  $(CC) $(CCFLAGS) $(CPPFLAGS) -c -o $@ $<
  ```

- 模式规则（Pattern Rules）：这种规则更加通用，因为可以利用模式规则定义更加复杂的依赖规则。模式规则看起来非常类似于后缀规则，但在目标名称的前面多了一个“%”号，同时可用于定义目标和依赖文件之间的关系，例如下面的模式规则定义了如何将任意一个 X.c 文件转换为 X.o 文件：

  ```
  %.c:%.o
  $(CC)$(CCFLAGS)$(CPPFLAGS)-c-o$@$<
  ```

以上是 MakeFile 文件的大体编写规范，对于比较复杂的软件包，要自己编写 MakeFile 文件也是一件令人烦恼的事。一般来说，这时软件提供商会提供一个 MakeFile 的示例文件，在这个文件的基础上按照自己的要求进行相应的修改就容易多了。另外，也可以使用

automake 和 autoconf 软件来生成 MakeFile 文件，具体使用方法请查看帮助文件。

2.2.4 debug 工具 GDB

在编写程序时，不可能保证程序代码是完全正确的，这时调试工具就是必需的。同 DOS 下有 debug 一样，在 Linux 操作系统中最常用的调试工具是 GDB。GDB 是 GNU 的一个重要软件，最早由 Richard Stallman 编写。

使用 GDB 的另一大好处是 GDB 支持嵌入式软件的开发模式——交叉调试，当运行 gdb 的 Linux 平台（宿主机）通过串行端口（或网络连接，或是其他的方式）连接到目标板时，gdb 可以对运行在目标板上的应用程序进行调试。由于这个特性，许多新开发的嵌入式操作系统都把 GDB 移植到其上作为调试工具。

GDB 能够观察另一个程序在执行时的内部活动，或程序出错时发生了什么？GDB 的主要功能有以下几点：

- 设置运行环境和参数，运行指定程序。
- 让程序在指定条件下停止和运行。
- 在程序运行停止后，检查变量、内存或寄存器的值，查看程序运行情况。
- 修改正在调试的程序的源代码，这样可以在线修正某个 bug 引起的问题，然后继续查找下一个 bug。

GDB 的使用可以直接在 shell 命令行下输入 gdb 并按回车键，再在 gdb 命令行下指定要调试的程序；也可以用 gdb filename 在启动时指定要调试的程序名。如果正常启动，屏幕将出现类似于以下的信息，并进入 GDB 命令模式：

GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)

Copyright 2001 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" fordetails.

This GDB was configured as "i386-redhat-linux".

(gdb)

GDB 可以运行在许多模式下，这些模式是在 GDB 运行时在命令行作为选项指定的。下面将对这些模式进行相应的说明。

- -nx 或-n：不执行任何初始化文件中的命令（一般 GDB 的初始化文件名.gdbinit）。一般情况下在这个文件中的命令会将所有的命令行参数传给 GDB 后执行。
- -quiet 或-q：安静模式。不输出上面显示的介绍和版本信息。这些信息在“批处理”中也将被跳过。
- -batch：批处理模式。当在批处理命令文件中的所有命令都被执行后，GDB 将返回状态 0；如果执行过程出错，将返回非 0 值。



- -cd DIRECTORY：把 DIRECTORY 作为 GDB 的工作目录，这时工作目录不再是当前目录（在一般情况下，GDB 默认把当前目录作为工作目录）。
- -b BIT/S：为远程调试设置串口波特率。
- -tty 设备名：使用其他设备作为程序的标准输入输出。这种模式对于嵌入式交叉调试很有用。

在 GDB 启动后，就进入 GDB 命令方式，这时就可以使用 GDB 的各种命令进行调试了。下面对它的各种调试命令进行详细说明。

为了使 GDB 能够正常工作，必须使程序在编译时包含调试信息。具体的调试信息包括程序里的每个变量的类型、在可执行文件里的地址映射以及源代码的行号等。如果没有这些信息，GDB 就默认到 init.c 中，这时就无法调试。若要使编译时包含这些信息，只需在使用 gcc 时加-g 选项即可。

下面对常用的 GDB 命令加以说明。

- 载入程序命令：file

在 GDB 内，载入程序很简单，使用 file 命令。如要加载 hello 程序用 file hello。当然，程序的路径名要正确。

- 退出 GDB 命令：quit

在 GDB 的命令方式下，输入 quit，就可以退出 GDB。也可以输入'C-d'来退出 GDB。

- 运行程序命令：run

当在 GDB 中已将要调试的程序载入后，可以用 run 命令来执行。如果程序需要参数，可以在 run 指令后接着输入参数，就像在 Shell 下执行一个需要参数的命令一样。

- 查看程序信息命令：info

info 指令用来查看程序的信息，它的参数非常多，但大部分不常用。一般用 info 指令最多的是用它来查看断点信息：info br，这时可以得到所设置的所有断点的详细信息，包括断点号、类型、状态、内存地址、断点在源程序中的位置等。info source 可以查看当前源程序。

- 列出源程序命令：list

这个命令从头开始将列出源程序代码，重复使用这个命令会接着前一次继续显示。若要列出某个指定函数：list FUNCTION。若以当前源文件的某行为中间显示一段源程序 list LINENUM，将显示另一个文件的一段程序：list FILENAME:FUNCTION 或 list FILENAME:LINENUM。

- 设置断点命令：break

这是最常用和最重要的命令，无论何时，只要程序已被载入，并且当前没有正在运行，就能设置、修改、删除断点。设置断点的命令是 break。有许多种设置断点的方法，如在函数入口设置断点：break FUNCTION；在当前源文件的某一行上设置断点：break LINENUM；在另一个源文件的某一行上设置断点：break FILENAME:LINENUM；在某个地址上设置断点，当调试的程序没有源程序时，可以用 break *ADDRESS。除此之外，设置一个断点，让它只有在某些特定的条件成立时程序才会停下来，可以称其为条件断点，它的命令格式为 break...if COND。COND 是一个布尔条件表达式，语法与 C 语言中的一样。条件断点与

一般的断点不同之处是每当程序执行到断点处，都要计算条件表达式，如果为真，程序才会停下来，否则程序会一直执行下去。

- 设置监视点命令：watch

当调试一个很大的程序，并且在跟踪一个关键的变量时，发现这个变量不知在哪儿被改动过，如何才能找到改动它的地方？这时可以使用 watch 命令。简单地说，监视点可以监视某个表达式或变量，当它被读或被写时让程序停下来。watch 命令的用法如下：

watch EXPRESSION

watch 指令是监视写操作的，当用户想监视某个表达式或变量的读操作的话，需要使用 rwatch 指令，具体用法是一样的。

- 显示表达式值的命令：print

最常用的检查数据的方法是：print exp，print 指令将打印 exp 表达式的值。默认情况下，表达式的值的打印格式依赖于它的数据类型。但可以用一个参数/F 来选择输出的打印格式。表达式 exp 中的变量必须是全局变量或当前堆栈区可见的变量。否则 GDB 会显示像下面的一条信息：

No symbol "varible" in current context

- 单步执行指令：step 或 next

单步执行指令有 step 和 next。step 可以跟踪进入一个函数，而 next 指令则不会进入函数。

- 继续执行命令：continue

当程序被停下来后，查看了所需的信息后，如希望程序执行下去，可输入 continue，这时程序将会继续执行下去。

- 产生可执行文件命令：make

通过 make 不用退出 GDB 就可以重新产生可执行文件。

- shell 命令

不离开 GDB 就可以执行 UNIX shell 命令。

在嵌入式 Linux 软件开发中，使用 GDB 交叉调试有两种方式：

- 目标机上的嵌入式 Linux 系统包含 GDB 工具。这时可以利用 TFTP 把在宿主机上开发的应用软件用 mount 命令挂载到目标机的一个开发目录下，再在主机上启动 minicom 超级终端，登录到目标机上。启动目标机的 GDB 程序，运行应用程序，之后就可以对应用程序进行调试了，调试的信息可以在 minicom 中看到。前面介绍的华恒目标板应用程序开发就属于这种方式。
- 目标板中不支持 GDB，这时就要在主机上运行 GDB，利用 GDB 的远程调试功能。在目标板上也要有一个名为 stub 的伺服程序，这个程序的作用是接受 GDB 的调试命令，解释执行，并按命令要求把调试结果返回给 GDB。例如，通过串口线的方式，在本地主机上输入 target remote /dev/ttyS0 命令，本地主机就可通过串口 1 和远程主机里面的 stub 程序相连接。当然，对于不同的体系结构的系统，需要编写不同的 stub 程序，在 GDB 的发布套件里面提供了默认 stub 文件，如针对 Sparc 机器的 sparc-stub.c 文件、针对 m68000 的 m68k-stub.c 和 intel 386 的 i386-stub.c 文件。



以上是 GDB 工具相关知识的介绍，在嵌入式 Linux 应用软件开发中，熟练地使用 GDB 是必要的，在以后的实例中也将继续介绍 GDB 在项目中的实际应用。

2.3 嵌入式 Linux 应用软件开发流程

从软件工程的角度来说，嵌入式应用软件也有一定的生命周期，如要进行需求分析、系统设计、代码编写、调试和维护等工作，软件工程的许多理论对它也是适用的。

但和其他通用软件相比，它的开发有许多独特之处：

- 在需求分析时，必须考虑硬件性能的影响，具体功能必须考虑由何种硬件实现。
- 在系统设计阶段，重点考虑的是任务的划分及其接口，而不是模块的划分。模块的划分则放到了任务设计阶段。
- 在调试时采用交叉调试方式。
- 软件调试完毕固化到嵌入式系统中后，它的后期维护工作较少。

下面将对它的开发流程进行详细说明。

2.3.1 对需求进行分析

对需求加以分析产生需求说明，需求说明过程给出系统功能需求，它包括以下内容：

- 系统所要实现的功能。
- 系统的输入、输出。
- 系统的外部接口需求（如用户界面）。
- 它的性能以及诸如文件/数据库安全等其他要求。

在实时系统中，常用状态变迁图来描述系统。在设计状态图时，应对系统运行过程进行详细考虑，尽量在状态图中列出所有系统状态，包括许多用户无需知道的内部状态，对于许多异常也应有相应处理。

此外，应清楚地说明人机接口，即操作员与系统间的相互作用。对于比较复杂的系统，形成一本操作手册是必要的，为用户提供使用该系统的操作步骤。为使系统说明更清楚，可以将状态变迁图与操作手册脚本结合起来。下面将以一个实例加以说明，引起状态变迁的用户操作已经在状态图上加以说明。

在对需求进行分析，了解系统所要实现的功能的基础上，系统开发选用何种硬件、软件平台也就可以确定了。

对于硬件平台，要考虑的是微处理器的处理速度、内存空间的大小、外部扩展设备是否满足功能要求等。如微处理器对外部事件的响应速度是否满足系统实时性要求，它的稳定性如何，内存空间是否满足操作系统及应用软件的运行要求，对于要求网络功能的系统，是否扩展有以太网接口等。

对软件平台而言，操作系统是否支持实时性及支持的程度、对多任务的管理能力是否

支持前面选中的微处理器、网络功能是否满足系统要求以及开发环境是否完善等都是必须考虑的。

当然，不管选用何种硬软件平台，成本因素都是要考虑的，嵌入式 Linux 正是在这方面具有突出优势。

如图 2-4 所示为一个比较典型的嵌入式系统的状态变迁过程。

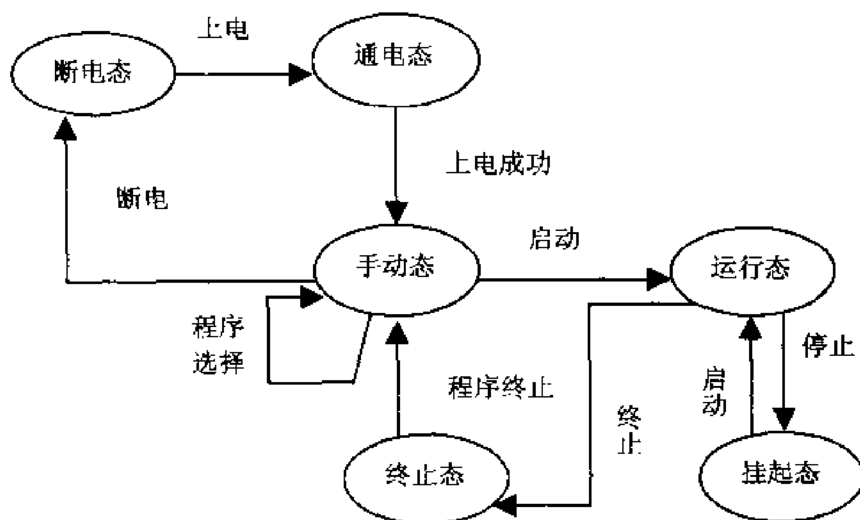


图 2-4　一个典型的系统状态变迁图

2.3.2　任务和模块的划分

在进行需求分析和明确系统功能后，就可以对系统进行任务划分。任务是代码运行的一个映像，是无限循环的一段代码。从系统的角度来看，任务是嵌入式系统中竞争系统资源的最小运行单元，任务可以使用或等待 CPU、I/O 设备和内存空间等系统资源。

在设计一个较为复杂的多任务应用系统时，进行合理的任务划分对系统的运行效率、实时性和吞吐量影响都极大。任务分解过细会不断地在各任务之间切换，而且任务之间的通信量也会很大，这样将会大大地加强系统的开销，影响系统效率。而任务分解过粗、不够彻底又会造成原来本来可以并行的操作只能按顺序串行执行，从而影响系统的吞吐量。为了达到系统效率和吞吐量之间的平衡和折中，在划分任务时应在数据流图的基础上，遵循下列步骤和原则。

1. 进行数据流分析

在系统需求分析的基础上，以数据流图作为分析工具。首先，从系统的功能需求开始分析系统中的数据流，分析数据在各状态转换之间的作用。然后，扩展数据流图，并分解到足够的深度，识别出主要的子系统和每个子系统的主要成分。



2. 划分任务

识别出系统的所有功能和它们之间的数据流后，下一步是要判断哪些操作是并行，哪些是串行，以划分任务。

在将一个软件系统分解为并行任务时，主要考虑的是系统内功能的异步性。这需要分析数据流图中的各功能变换，确定哪些变换可以并行，而哪些在本质上又是顺序的。一般并行的功能变换应属于不同的任务，而串行的可以属于同一任务。任务的划分包括确定哪些变换属于哪个任务，及确定各任务的优先级。它们的划分原则如下：

（1）I/O 依赖性。

如果功能变换依赖 I/O，那么它的运行速度常常受限于与它互操作的 I/O 设备的速度。在这种条件下，功能变换应单独成为一个任务。

（2）功能的时间关键性。

具有时间关键性的功能需要以高优先级运行，因此不能把它加到其他任务中运行，应成为一个独立的高优先级任务。

（3）计算需求。

需要进行大量计算但又不具有时间紧迫性的功能或功能集合，可以作为较低优先级的任务运行，以消耗 CPU 的剩余时间。

（4）功能内聚。

完成的功能紧密相关的变换可以组成一个任务，因为这些功能间的数据通信较多，把它们作为一个个独立的任务反而会增加系统开销。反之，把每个变换作为同一任务中一个独立模块，不仅保证了模块级的功能内聚，而且保证了任务级的功能内聚。

（5）周期执行。

一个需要周期执行的变换可以作为一个独立的任务，按一定的时间间隔被激活。

3. 定义任务接口

在划分好任务之后，要确定任务间的接口。在数据流图中，接口是以数据流或数据存储区的形式存在，在这里要把它们具体化下来，确定将采用何种格式的接口。

通常由两种任务接口模块来处理接口问题，即任务间通信模块和任务同步互斥模块，这些模块对调用它的任务来说一般是操作系统级的任务调用。

任务通信模块处理任务间的所有通信情况。一般它会定义一个数据结构，并定义对该数据结构的访问过程，如对消息队列、管道等结构的访问。任务通信模块总是运行在调用它的任务中，因而，它有可能在两个任务中并发执行，所以在访问过程中必须提供必要的同步和互斥条件来确保数据的一致性和正确性。

任务同步互斥模块是当任务之间不需要传送真正的信息时使用的，它用事件来实现同步目的。目标任务等待一个或几个事件的发生，源任务发送事件信号激活目标任务。

以上对任务的划分进行了说明，在设计一个复杂应用时，上述的划分原则仅能作为一个初步参考，真正的设计还需要详细分析，才能使系统达到预定的效率和吞吐率。

设计好任务后，就可以在任务内进行模块划分，确定各模块功能和接口，这和普通的软件设计一样，软件工程的方法在此也是适用的。

2.3.3 生成代码

应用软件的生成又可以分为 3 个阶段：源代码的编写，将源代码编译成各个目标模块，将所有目标模块及相关库文件链接成可供下载调试或固化的可执行程序。具体嵌入式应用软件的生成过程如图 2-5 所示。

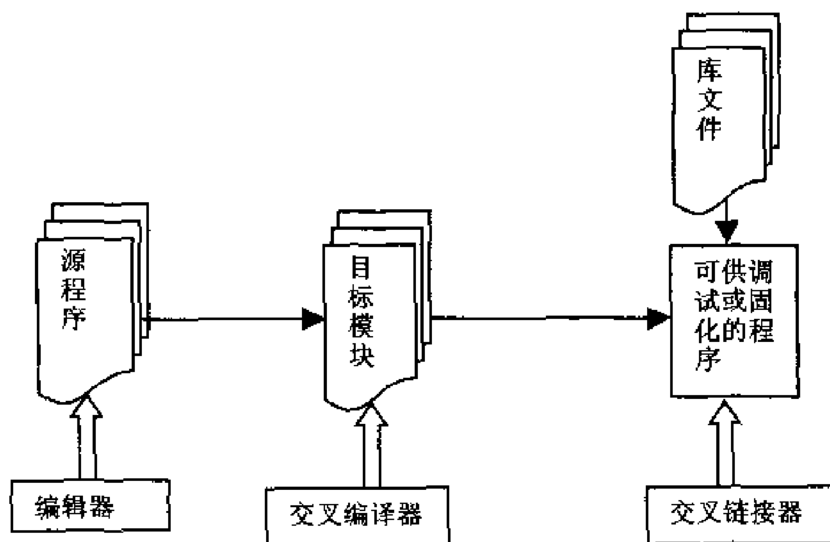


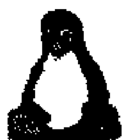
图 2-5 嵌入式应用软件的生成

从表面看，这一过程似乎与普通的计算机软件开发过程一样，但其实它们有本质的区别，关键就在于编译器和链接器上。编译器要将源程序编译成对应特定目标处理器的目标代码，因此称为交叉编译器。不同的目标处理器所对应的编译器不尽相同，为了提高编译质量，有许多硬件厂商还会针对自己开发的处理器的特性定制编译器，既提供高级编程语言的支持，又能很好地对目标代码进行优化。现在许多商用的嵌入式操作系统也会提供对多种处理器的支持。

应用程序的运行方式有两种：调试方式和固化方式，不同方式下程序代码或数据在目标机内存中的定位不同。宿主机上的嵌入式开发系统，一般会提供一定的工具和手段对目标应用程序的运行方式和内存定位加以选择和配置，链接器再根据这些配置信息将目标模块和库文件中的模块链接成可执行程序，因此称为交叉链接器。同目标模块相链接的运行库也是与嵌入式应用相关的，如在 uClinux 中使用的库是 uClibc 库，与标准 Linux 库 libc 是有差别的。

2.3.4 调试代码

调试嵌入式 Linux 应用软件主要使用 GDB 远程调试功能，利用 GDB 进行远程调试并



不像在本机上调试一个可执行程序那么简单，因为需要在两台已经连接的机器的基础上加以调试。

在 GDB 里面有一个调试目标（Target）的概念。调试目标就是程序所获得的执行环境。一般情况下，要调试的程序和当前所在的环境是完全一样的，那么使用 file 或者 core 命令可以指定调试目标（file 命令用于指定用来调试的可执行文件，core 命令用于指定调试的时候需要导入的 core dump 文件）。另外一种情况是，如果需要调试的程序和 GDB 所运行的环境不同，或者说需要调试的环境上根本无法运行 GDB，那么就没有办法使用 file 或者 core 命令来指定调试目标了。这时，就需要使用远程调试功能，利用一台可以使用 GDB 的主机，通过串口的通信协议和被调试的程序所在的目标机加以连接，从而调试程序。在 GDB 里面是使用 target 命令完成这项工作的。

指定需要调试的远程机器的方法是使用 target remote 命令，后面紧接着需要和远程机器连接的设备，如/dev/ttyS0（串口 1）等。在 GDB 里面内嵌有串口的通信协议，并且规定了和远程机器的调试命令的一些数据传输包的格式。

在远程目标机上，需要实现一个 stub 文件，在这个文件里面需要提供串口连接的协议和传送数据信息的方法。可以这样说，stub 文件代替了在本地主机上 GDB 串口协议的位置，从而实现两台机器的连接，同时 stub 文件监控被调试的程序，把调试信息显示给主机。

GDB 远程调试的原理如图 2-6 所示。

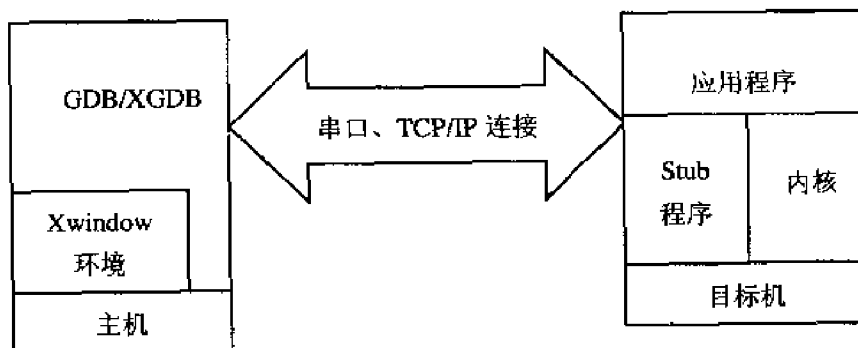


图 2-6　GDB 远程调试环境原理图

2.3.5　固化运行

在调试完成确定运行无误后，就可以固化运行了，这时应用程序将完全烧入目标板的非易失性存储器（如 ROM、FLASH）中，并且在真实的硬件环境中运行。与调试阶段的代码相比，固化的代码运行有如下两个差别：

（1）代码定位不同。

固化运行时，在去掉调试监控模块和加入 Boot 模块的情况下，要重新设置定位控制文件，则集成编译工具将按照应用各代码模块的属性把它们定位到相应的目标板存储空间中去。

在嵌入式系统中，一般使用两种存储器：一种是可读写的 RAM，另一种是非易失性的存储器，如 ROM、FLASH Memory 等。这两种存储器同时被映射到系统的寻址空间。一般

RAM 被映射到地址空间的低端，而 ROM、FLASH Memory 等被映射到地址空间的高端。

在调试方式下，全部应用代码和数据都定位于 RAM 中，代码在 RAM 中运行；在固化方式下，代码和数据都存储在非易失性存储器中，系统启动时要先将数据移到 RAM 中，而程序代码既可以在 ROM、FLASH Memory 中运行，也可以到 RAM 中运行。

（2）初始化部分不同。

在固化运行时，要创建 Boot 模块，此模块将在启动时作为整个应用系统代码的入口模块。当应用程序在真实的目标环境下运行时将首先执行该程序，完成对 CPU 环境的最初初始化。而在调试阶段，Boot 模块的一些功能由目标板上的监控器程序实现。在嵌入式环境中，Boot 模块一般包含以下几个功能块：

- 初始化芯片的引脚，即按照系统的最终配置定义处理器芯片引脚功能。
- 初始化一些系统外部控制寄存器，如 WatchDog、DMA、时钟计数器、中断控制寄存器等。
- 初始化基本的输入/输出设备，在嵌入式系统中一般为串口或并口。
- 执行数据复制，将一些存储在非易失性存储空间的数据复制到真实的运行空间中去。
- 若有 MMU，将初始化 MMU，包括片选控制器等。

完成这些工作后，就可以用开发环境提供的固化工具把应用程序固化到目标机的 ROM、FLASH Memory、启动软盘或硬盘中，当重新启动目标机后，应用程序就会自动装入运行。

至此，嵌入式应用软件的开发工作就告一段落了。但在实际开发过程中，经常会碰到调试好的程序固化后却又不正常的情况，针对这种情况，一些功能强大的开发环境会提供调试固化程序的工具，借助此工具可以查找到问题的所在。

2.4 一个简单的应用程序——Hello World

在本章前面已经对嵌入式 Linux 开发平台、开发工具和开发流程进行了详细说明，在本节中将通过实例对前面介绍的知识加以总结和巩固。

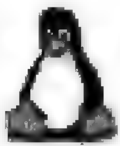
本实例所实现的功能相当简单，就是把一字符串按它的反序输出，例如"Hello embedded Linux world!"按它的反方向输出。

首先，在主机上用 vi 或 Emacs 编写如下源代码（/demo/chap2/2-1/test1.c）：

```
#include <stdio.h>

void my_print(char *string);
void my_print2(char *string);

main()
{
```




```
        char my_string[]="Hello embedded Linux world";

        my_print(my_string);
        my_print2(my_string);
}

void my_print(char *string)
{
        printf("The string is %s\n",string);
}

void my_print2(char *string)
{
        char *string2;
        int size,i;

        size=strlen(string);
        string2=(char *)malloc(size+1);
        for(i=0;i<size;i++) {
              string2[size-i]=string[i];
        }
        string2[size+1]='\0';
        printf("The string printed backward is %s\n",string2);
}
```

下面就是对源代码进行编译生成可执行文件，可以通过两种方式实现，一种是直接用 gcc 程序编译连接，自己生成并维护可执行文件；另一种是使用 make 程序，利用 MakeFile 脚本文件生成。

若是比较简单的程序，维护起来不是很复杂，可以使用第一种方式，在本程序中它的用法是：gcc -g -o hello hello.c。

注意：加上-g 选项是用来产生调试信息的。

若是第二种方式，可以直接编写 MakeFile 文件，也可以对现有的 MakeFile 文件作一些修改即可。如在华恒的开发板上，可以复制/LinuxPPC/usr/src/application/下某个目录中的 MakeFile 到 hello 目录下。如果以 application/FLASH 的 MakeFile 为模板，修改 MakeFile 的 4 处 "put" 内容为 "hello" 即可，修改后的内容如下：

```
SOURCES = hello.c
OBJS = hello.o
#$(SOURCES:.c=.o)
```

```
.PHONY: clean mrclean distclean depend all dist
hello: ${OBJS}
${CC} -o $@ hello.o ${LDFLAGS} ${LIBS}
```

其他部分可以不作修改。文件中的几个宏在 make 文件里有默认定义，然后执行 make，在 hello 目录下将生成可执行文件 hello。

下面开始对应用程序进行调试，还是在华恒 HHPPC860-3COM-2ETH-R1 套件上进行。

首先，将主机跟目标机用串口线或网线连接起来，把编译通过后的可执行程序 hello 复制到宿主机根目录下以方便下面对 mount 执行，避免 mount 宿主机后要进入很深的目录；然后，再在宿主机上启动 minicom 作为目标板的仿真终端；最后，在 mount 宿主机上存放该应用程序的目录，例如：mount –o nolock 192.168.1.199:/ /mnt，其中 192.168.1.199 是主机的 IP 地址，在 minicom 下执行：

cd /mnt

./hello

调试信息将通过串口打印到宿主机的 minicom 屏幕上，这样便可进行应用程序的调试。若有问题，则切换到编辑编译，只要不重新启动目标端就不必进行任何其他操作，因为 mount 的宿主机硬盘上的应用程序会自动覆盖更新，再重新执行的就是更改后的新版本。调试过程就是这样反复调试、更改、编译再调试，直至程序工作正常，而不必烧写板了。

如果执行上面代码的 hello 程序，会看到如下结果：

The string is Hello embedded Linux world!

The string printed backward is

输出的第一行是正确的，但第二行打印出来的内容并不是所期望的。设想的输出应当是：

The string printed backwars is !dlrow xuniL deddebme olleH

由于某些原因，导致 my_print2()函数没有正常工作，这时需要用 GDB 来调试查看到底是什么地方出了问题？

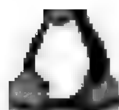
在 hello 程序的目录下输入：gdb hello。这时就已经启动 GDB，开始调试 hello 程序了，也可以在 GDB 的命令提示符下输入 file 命令来载入要调试的程序，

(gdb)file test

这时可以 GDB 的 run 命令来运行 test 程序，可以看到它的输出结果和在外面直接运行是一样的。问题是为什么打印反序字符串的函数没有正常工作？为了找出症结所在，在 my_print2()函数的 for 语句后设一个断点，具体做法是在 GDB 命令提示符下输入 list 命令数次，列出源代码。

注意：在 GDB 提示符下按回车键将重复上一命令。

```
(gdb)list
1        #include    <stdio.h>
2
3        void my_print(char *string);
4        void my_print2(char *string);
```




```
5
6           main()
7           {
8                  char my_string[]="Hello embedded Linux world";
9
10                 my_print(my_string);
(gdb)list
11                 my_print2(my_string);
12                 }
13
14          void my_print(char *string)
15          {
16                 printf("The string is &s\n",string);
17          }
18
19          void my_print2(char *string)
20          {
(gdb)list
21                 char *string2;
22                 int size,i;
23
24                 size=strlen(string);
25                 string2=(char *)malloc(size+1);
26                 for(i=0;i<size;i++) {
27                 string2[size-i]=string[i];
28                 }
29                 string2[size+1]='\0';
printf("The string printed backward is %s\n",string2);
(gdb)list
31          }
(gdb)
```

根据列出的源代码，可以看出要设断点的地方在第 27 行，在 GDB 命令行下输入如下命令设置断点：

```
(gdb)break 27
Breakpoint 1 at 0x8048510:file hello.c,line 27.
(gdb)
```

现在再输入 run 命令，将产生如下的输出：

```
(gdb)run
Starting program: /mnt/hello
The string is Hello embedded Linux world!
```

Breakpoint 1, my_print2(string=0xbffff960 "Hello embedded Linux world!") at hello.c: 27

string2[size-i]=string[i];

(gdb)

通过设置一个 string2[size-i]变量值的观察点，可看出错误是怎样产生的，做法是输入 watch 命令：

(gdb)watch string2[size-i]

Hardware watchpoint 2: string2[size-i]

(gdb)

现在可以用 next 一步步地执行 for 循环了，经过第一次循环后，GDB 将告诉用户 string2[size-i]的值是 H，这是正确的，后来的数次循环都是对的，当 i=26 时，表达式 string2[size-i]的值为 'l'，size-i 的值为 1，最后一个字符都已经复制到新字符串里去了。如果再执行下一步循环，会看到已经没有值分配给 string2[0]了，而它是一个新字符串的第一个字符，因为 malloc()函数在分配内存时已把它们初始化为空字符（null），所以 string2 的第一个字符为空字符，这样整个字符串都被认为是空字符，故反序没有输出。

现在找到问题了，修正这个错误很容易，应该把代码里写入 string2 的第一字符的偏移量改为 size-1 而不是 size。以下是修改后的 test1.c 的源代码（/demo/chap2/2-1/test2.c）：

```
#include <stdio.h>

void my_print(char *string);
void my_print2(char *string);

main()
{
        char my_string[]="Hello embedded Linux world";

        my_print(my_string);
        my_print2(my_string);
}

void my_print(char *string)
{
        printf("The string is %s\n",string);
}

void my_print2(char *string)
{
        char *string2;
        int size,i;
```




```
        size=strlen(string);
        string2=(char *)malloc(size+1);
        for(i=0;i<size;i++) {
            string2[size-i-1]=string[i];
        }
        string2[size+1]='\0';
        printf("The string printed backward is %s\n",string2);
}
```

在调试完成后，就可以下载烧制到 ROM 或 FLASH 中运行了。下面还是以华恒的开发板为例加以说明。

在华恒开发板的应用程序开发上，可以把应用程序和 Linux 内核一起编译再烧制到 FLASH 中，具体做法是：在/LinuxPPC/usr/src/linux-2.4.4-2002-02-14/arch/ppc/mbxboot 目录下有一个文件 amdisk.image.gz，ramdisk.image.gz 为 Linux 的文件系统映像压缩文件。用户可以在文件系统中加入自己的应用，例如，可以将 ramdisk.image.gz 复制到根目录下，新建一个 ramdisk 目录并解开 ramdisk.image.gz：

cp ramdisk.image.gz /

cd /

mkdir ramdisk

gunzip /ramdisk.image.gz

在根目录下会生成 ramdisk.image，ramdisk.image 为解开后的 Linux 的文件系统映像文件。再将 ramdisk.image 文件系统映像文件 mount 到新建目录 ramdisk 中：

mount -o loop ramdisk.image ramdisk/

cd ramdisk/

查看目录中的内容如下：

bin etc linuxrc mnt sbin usr

dev lib lost+found proc tmp var

它完全就是 Linux 的文件系统（与目标板启动后的文件系统完全一致）。此时用户可以在其中加入自己的应用程序。下面举例说明如何将 2.4 节开始部分生成的 hello 加入到文件系统中。

cd /ramdisk

mkdir application（新建目录名可以自己定义）

cd application

mkdir hello

cd hello

cp /hello（复制 hello 到 hello 目录下，注意此时生成的可执行文件 hello 在“/”目录下，当然用户可指定自己的应用程序路径）。

cd /

umount /ramdisk

压缩生成新的 ramdisk.image 文件系统映像文件：

gzip /ramdisk.image /ramdisk.image.gz

复制 ramdisk.image.gz 到/LinuxPPC/usr/src/linux-2.4.4-2002-02-14/arch/ppc/mbxboot/目录下：

cp /ramdisk.image.gz /LinuxPPC/usr/src/linux-2.4.4-2002-02-14/arch/ppc/mbxboot/

重新编译 Linux 内核：

cd /LinuxPPC/usr/src/linux/

./easy_build

生成 Linux 内核映像文件 linux..bin，然后烧写 linux.bin 到 FLASH 中。

启动目标板可以看到它的目录结构：

application bin etc linuxrc mnt sbin usr

dev lib lost+found proc tmp var

这时，在文件系统中出现了 application 目录。

cd application

cd hello

在 hello 目录中出现了可执行文件 hello，这时就可以运行该应用程序了。方法为：

./hello

至此，整个开发过程就已完成，以上开发过程的细节若有不明之处可与华恒厂商联系，寻求技术支持。

2.5 小结

在本章里，系统地学习了在嵌入式 Linux 平台上开发应用程序的基本方法。首先，对嵌入式 Linux 平台进行了说明，介绍了比较典型的嵌入式 Linux——uClinux 及其硬件平台，并以华恒开发板为例，对开发平台的建立过程进行了说明。重点是要掌握嵌入式应用开发平台的特点及其与普通软件开发平台的区别。

其次，还学习了嵌入式 Linux 应用软件开发工具的相关知识。其中包括 vi 编辑器、gcc 编译器、MakeFile 文件的编写和 GDB 在嵌入式 Linux 开发中的应用，这些都是实际开发中常用的工具，熟练掌握可以大大提高工作效率。

另外，还介绍了嵌入式应用的开发流程，包括需求分析、任务划分、代码编写、调试和固化的整个过程。对嵌入式应用开发比较陌生的读者来说，了解整个开发流程是十分必要的。

最后通过具体的实例，对本章所介绍的重点知识作了更进一步的总结和说明。

2.6 思考题

1．嵌入式开发平台有何特点？完整的开发平台包括哪些东西？



2. 请简述 uClinux 及其硬件平台 uCsimm 的特点。

3. 若有条件请按本章介绍的方法在华恒开发板上建立一个嵌入式 Linux 开发平台。

4. 使用 vi 编辑器如何进行查找、替换、复制和粘贴等操作？

5. 请在 Linux 平台下对本章中介绍的 hello 例程进行编译，并按书中的调试步骤对原来的错误代码进行调试，以熟练掌握 GDB 调试方式。

6. 简述嵌入式软件开发流程。

7. 任务划分的原则是什么？如何定义任务接口？

第二篇 系 统 篇

第3章 任 务 管 理

知识点：

- 任务的基本概念
- 任务状态及其转变
- 任务调度、目标和算法
- Linux 任务管理 API

本章导读：

任务管理是嵌入式实时操作系统中最重要的部分。与传统的分时操作系统不同，在嵌入式实时操作系统中，应用程序开发的基本单位是任务（Task）。本章将详细介绍嵌入式系统中任务的基本概念、任务状态转变和任务调度等方面的内容。



3.1 任务概述

任务可理解为一个具有独立功能的无限循环的程序段的一次运行活动，其具有动态性、并行性、独立性、异步性、可再现性的特点。

- 动态性：是指任务的状态是不断变化的。嵌入式系统中，任务状态一般可分为：休眠态（dormant）、就绪态（ready）、运行态（running）、挂起态（suspended）和睡眠态（sleep）等。
- 并行性：是指在操作系统支持下，多个任务在宏观上并行，微观上串行执行（多处理器系统中，也存在微观上的并行）的特点。
- 独立性：是指任务是一个独立运行、调度、申请资源的基本单位，一段程序必须成为一个任务才能投入运行。
- 异步性：是指任务相互对立，以不可预知的速度运行，操作系统必须提供相应机制协调各个任务的运行。
- 可再现性：是指任务在相同的初始条件和环境，多次运行应该得到相同的结果。

根据嵌入式实时操作系统开发者的意图，任务可以实现为进程或线程。在 Vxworks、Psos 等传统嵌入式实时系统中，任务可以看作是一个线程，一个任务可以由一个线程控制块 TCB 和一个线程执行体构成。为了提高系统的实时性，所有线程都使用同一地址空间，即所谓的“平面保护模式”，在这种模式下，线程之间没有保护，任务 A 可以访问任务 B 的地址空间，甚至是 A 的私有数据结构。“平面保护模式”的好处在于：所有任务处于同一地址空间，任务切换不需要进行地址空间的切换，并且任务间通信、任务间同步机制和内存管理机制等实现相对容易，从而可以简化系统的实现，提高系统的实时性能。

但是，这种“平面保护模式”实质上是一种无保护模式，某个应用程序出错，可能导致整个系统的崩溃。这种缺陷对于某些安全关键的系统，如核电站控制系统、航空航天导航系统、生命监护系统等，是不可容忍的。因此，一些新型嵌入式操作系统，在任务管理中引入了地址空间隔离机制。在这种系统中，单个任务出错，并不会影响其他的任务运行。加拿大 QNX 软件系统有限公司推出的 QNX 嵌入式实时操作系统就是这种系统的典型代表。

3.1.1 标准 Linux 进程

在 Linux 中，任务被映射为进程。Linux 的进程模型源自 UNIX 系统，使用 Fork 系统调用可以创建一个与调用进程相同的复制。调用 Fork()函数的进程叫做父进程，新创建的进程叫做子进程，父进程和子进程一起并行运行。父进程可以派生出更多的子进程，而子进程也可以派生出新的子进程。这样，从初始进程开始，随着进程树的不断繁殖、演化，就得到了一个动态的进程树。初始进程好比是进程树的树根，初始进程的子进程好比是进程树的叶节点，在 Linux 中，进程树的深度只受最大进程数的限制。

为了说明 Linux 中进程树的表示方法，首先介绍 Linux 的进程控制块。Linux 中的每一

个进程都由一个 task_struct 数据结构来描述，task_struct 就是通常所说的进程控制块，即 PCB。task_struct 存储了管理任务所需要的大量信息，包括与进程树表示相关的部分，其结构如下所示（摘自 linux\linux2.4.x\include\linux\sched.h 文件）：

```
struct task_struct {
        ...
        struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
        ...
}
```

其中，5 个数据域都是 task_struct 结构的指针，分别代表：

- p_opptr：进程的祖先节点。
- p_pptr：进程的父节点。
- p_cptr：进程的子节点。
- p_ysptr：进程的右兄弟。
- p_osptr：进程的左兄弟。

3.1.2 任务的数据结构表示

在不同类型的操作系统中，任务的数据结构表示不尽相同。在 UNIX 类操作系统中，从任务静态来看就是一个存放在外存里的可执行程序文件，当操作系统装载该程序运行时，将建立任务的动态数据表示。例如，在 UNIX 操作系统中，一个进程代表一个任务，它主要由代码段、数据段、未初始化数据（BSS）段和堆栈段等组成。

3.1.3 实时任务

实时任务是指对任务的开始、运行和结束等都有着时间特殊要求的任务。实时任务往往要求在某个时限到来前完成运行。实时任务可以是周期的、间发的或非周期的。T_i 是周期任务，是指 T_i 每隔 P_i 秒释放一次，P_i 是任务 T_i 的周期。周期限制要求任务在每个周期内运行，且仅运行一次。通常，任务的周期也是任务的时限。间发任务是指任务释放的时间间隔不规则，但时间间隔存在上限，即任务触发的最大时间间隔确定。因此，若 t 秒为 T_i 的最大时间间隔，在 t 秒内 T_i 至少触发两次。有时，间发任务也被看成为周期任务。

3.1.4 嵌入式 Linux 中的进程

大多数的嵌入式 Linux 版本，保留了标准 Linux 的进程模型。少数嵌入式 Linux 版本



对标准 Linux 的进程模型进行了更改。如 uClinux 为了去掉 Linux 的虚拟内存管理，将标准的 Fork 系统调用改为了 Vfork；而在嵌入式强实时 RTLinux 中，则将 Linux 扩展成了双内核结构，即实时内核和 Linux 内核（非实时），而其任务管理也相应地分成了实时任务管理和非实时任务管理。

3.2 任务状态的转变

任务在运行中不断地改变其运行状态，通常一个运行的进程必须具有以下 3 种基本状态：

- 运行态：任务已经获得处理器，正在执行。
- 就绪态：任务满足运行条件，等待获得处理器。
- 阻塞态：因任务的运行条件尚不满足，而暂时停止执行的状态。

图 3-1 显示了任务状态转换过程。图中的 4 个箭头线表明了 4 种可能的转换。

- 就绪态－运行态

处于就绪态的任务，当调度器为之分配处理器后，转变为运行态，开始执行。

- 运行态－就绪态

正在执行的任务，在占用足够多的处理器时间后，调度器决定让其他的任务占用处理器时，将被剥夺执行，而转变为就绪态。比如在实时系统中，使用时间片轮转调度算法时，任务会因为已将自己拥有的时间片用完，而发生这种转变。

- 运行态－阻塞态

正在执行的任务，因为发生某种事件而无法继续执行时，将放弃处理器，进入阻塞态。例如，任务请求被其他任务占用的临界资源，或是任务调用系统阻塞原语请求将自己挂起。

- 阻塞态－就绪态

处于阻塞态的任务，当获取等待资源，或是等待的外部事件到达时，将转变为就绪态，准备重新开始运行。

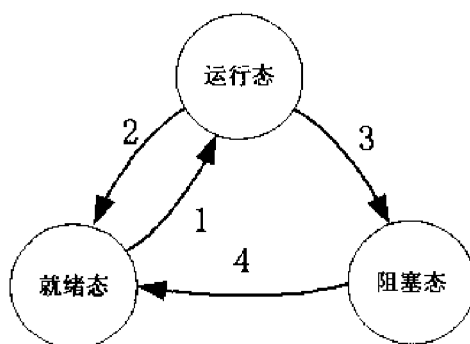


图 3-1 任务状态转换图

以上只是任务的 3 个基本状态，在实际的操作系统中，为了方便系统的管理和实现，任务被进一步地划分为更多的状态，如新状态和退出状态等。在 Linux 2.4.x 中，任务有以

下几种状态（摘自 linux\linux2.4.x\include\linux\sched.h 文件）：

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_ZOMBIE             4
#define TASK_STOPPED            8
```

各种状态的含义分别是：

- TASK_RUNNING：正在运行的进程（是系统的当前进程）或准备运行的进程（在 Running 队列中，等待被安排到系统的 CPU）。处于该状态的进程按自己的调度规则与其他该状态的进程共享 CPU。
- TASK_INTERRUPTIBLE：处于等待队列中的进程，待资源有效时唤醒，也可被其他进程通过信号唤醒。
- TASK_UNINTERRUPTIBLE：处于等待队列中的进程，直接等待硬件条件，待资源有效时唤醒。与 TASK_INTERRUPTIBLE 不同的是，处于该状态的任务，不可以由其他进程通过信号唤醒。该状态主要用于设备驱动程序开发。
- TASK_ZOMBIE：终止的进程，是进程结束运行前的一个过渡状态（僵死状态）。虽然此时已经释放了内存、文件等资源，但是在 Task 向量表中仍有一个 task_struct 数据结构项。它不进行任何调度或状态转换，只是等待父进程将它彻底释放。
- TASK_STOPPED：进程被暂停，必须接收到其他进程的信号才能被唤醒。正在调试的进程可以处于该状态。

可见，Linux 的任务状态设计没有照搬教科书的理念，而是根据内核的特点进行了改进。在 Linux 中一个任务在其生命周期中的状态转变过程如图 3-2 所示。

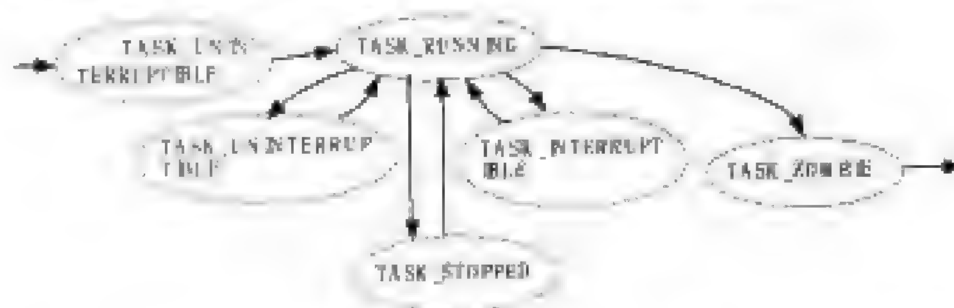
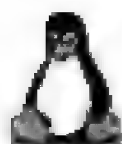


图 3-2　Linux 任务状态转换图

接下来，对照 Linux 的源代码介绍任务状态的转变过程。

首先，在进程被创建时，任务状态被设置为 TASK_UNINTERRUPTIBLE。

```
int do_fork(unsigned long clone_flags, unsigned long stack_start,
            struct pt_regs *regs, unsigned long stack_size)
```




```
{
        ...
        p->did_exec = 0;
        p->swappable = 0;
        p->state = TASK_UNINTERRUPTIBLE;
        copy_flags(clone_flags, p);
        p->pid = get_pid(clone_flags);
        ...
}
```

然后，在进程将要创建完成时，do_fork()函数调用 wake_up_process()函数，通过"唤醒"的方式将新创建的进程加入就绪队列，并将其状态改为 TASK_RUNNING（摘自 Linux\kernel\sched.c）。

```
static inline int try_to_wake_up(struct task_struct * p, int synchronous)
{
        unsigned long flags;
        int success = 0;

        /*
         * We want the common case fall through straight, thus the goto.
         */
        spin_lock_irqsave(&runqueue_lock, flags);
        p->state = TASK_RUNNING;
        if (task_on_runqueue(p))
                goto out;
        add_to_runqueue(p);
        if (!synchronous || !(p->cpus_allowed & (1 << smp_processor_id())))
                reschedule_idle(p);
        success = 1;
out:
        spin_unlock_irqrestore(&runqueue_lock, flags);
        return success;
}
```

任务在运行过程中，如果任务等待某个事件或资源，当等待的事件发生或者资源有效时，任务被唤醒。

另外，在任务等待过程中，如果收到信号或者被中断，则可能出现两种情况。如果该任务设置为不能被信号或中断唤醒，这种情况下，任务进入 TASK_ UNINTERRUPTIBLE 状态；反之，该任务进入 TASK_INTERRUPTIBLE 状态。

TASK_UNINTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE 两种状态，在 Linux 的设备驱动程序开发中被大量使用，另外，也用来实现可被打断的休眠（sleep）操作和可被打断的信号量获取（down）操作。接下来，将介绍 Linux 中两种休眠操作的实现方法（摘自 Linux\kernel\sched.c）。

```
#define   SLEEP_ON_VAR                          \
        unsigned long flags;                    \
        wait_queue_t wait;                      \
        init_waitqueue_entry(&wait, current);

#define   SLEEP_ON_HEAD                         \
        wq_write_lock_irqsave(&q->lock,flags);  \
        __add_wait_queue(q, &wait);             \
        wq_write_unlock(&q->lock);

#define   SLEEP_ON_TAIL                         \
        wq_write_lock_irq(&q->lock);            \
        __remove_wait_queue(q, &wait);          \
        wq_write_unlock_irqrestore(&q->lock,flags);

void interruptible_sleep_on(wait_queue_head_t *q)
{
        SLEEP_ON_VAR

        current->state = TASK_INTERRUPTIBLE;

        SLEEP_ON_HEAD
        schedule();
        SLEEP_ON_TAIL
}

void sleep_on(wait_queue_head_t *q)
{
        SLEEP_ON_VAR

        current->state = TASK_UNINTERRUPTIBLE;

        SLEEP_ON_HEAD
        schedule();
        SLEEP_ON_TAIL
}
```



任务收到 SIGSTOP、SIGSTP、SIGTTIN、SIGTTOU 等信号或者正在被其他进程调试时，任务处于 TASK_STOPPED 状态，典型用法是在系统调用 trace 中，任务状态被设为 TASK_STOPPED 状态。下面是该系统调用的实现代码（摘自 Linux\arch\i386\kernel\ptrace.c）：

```
asmlinkage void syscall_trace(void)
{
        if ((current->ptrace & (PT_PTRACED|PT_TRACESYS)) !=
                    (PT_PTRACED|PT_TRACESYS))
            return;
        /* the 0x80 provides a way for the tracing parent to distinguish
            between a syscall stop and SIGTRAP delivery */
        current->exit_code = SIGTRAP | ((current->ptrace & PT_TRACESYSGOOD)
                                    ? 0x80 : 0);
        current->state = TASK_STOPPED;
        notify_parent(current, SIGCHLD);
        schedule();
        /*
         * this isn't the same as continuing with a signal, but it will do
         * for normal use.   strace only continues with a signal if the
         * stopping signal is not SIGTRAP.   -brl
         */
        if (current->exit_code) {
              send_sig(current->exit_code, current, 1);
              current->exit_code = 0;
        }
}
```

任务在运行完毕，准备退出时，会将任务的状态设置为 TASK_ZOMBIE 状态，该功能由函数 exit_notify()实现。下面是该函数的关键代码（摘自 Linux\arch\i386\kernel\ptrace.c）：

```
static void exit_notify(void)
{
        struct task_struct * p, *t;

        forget_original_parent(current);

        t = current->p_pptr;

        if ((t->pgrp != current->pgrp) &&
              (t->session == current->session) &&
              will_become_orphaned_pgrp(current->pgrp, current) &&
```

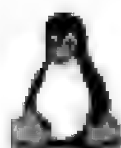
```
        has_stopped_jobs(current->pgrp)) {
        kill_pg(current->pgrp,SIGHUP,1);
        kill_pg(current->pgrp,SIGCONT,1);
}


if(current->exit_signal != SIGCHLD &&
    ( current->parent_exec_id != t->self_exec_id  ||
      current->self_exec_id != current->parent_exec_id)
    && !capable(CAP_KILL))
    current->exit_signal = SIGCHLD;

write_lock_irq(&tasklist_lock);
current->state = TASK_ZOMBIE;
do_notify_parent(current, current->exit_signal);
while (current->p_cptr != NULL) {
    p = current->p_cptr;
    current->p_cptr = p->p_osptr;
    p->p_ysptr = NULL;
    p->ptrace = 0;

    p->p_pptr = p->p_opptr;
    p->p_osptr = p->p_pptr->p_cptr;
    if (p->p_osptr)
        p->p_osptr->p_ysptr = p;
    p->p_pptr->p_cptr = p;
    if (p->state == TASK_ZOMBIE)
        do_notify_parent(p, p->exit_signal);
    /*
     * process group orphan check
     * Case ii: Our child is in a different pgrp
     * than we are, and it was the only connection
     * outside, so the child pgrp is now orphaned.
     */
    if ((p->pgrp != current->pgrp) &&
        (p->session == current->session)) {
        int pgrp = p->pgrp;

        write_unlock_irq(&tasklist_lock);
        if (is_orphaned_pgrp(pgrp) && has_stopped_jobs(pgrp)) {
            kill_pg(pgrp,SIGHUP,1);
            kill_pg(pgrp,SIGCONT,1);
```




```
            }
            write_lock_irq(&tasklist_lock);
        }
    }
    write_unlock_irq(&tasklist_lock);
}
```

3.3 任务调度

3.3.1 调度目标

任务调度的主要目的是合理地分配处理器资源，具体来说，调度就是从当前就绪的任务中选择一个投入运行。操作系统中实现该功能的部分称做调度器（scheduler），调度器使用的算法就称为调度算法。

调度是实现多任务并发执行的必要手段，不同类型的操作系统有着不同的调度目标。在传统的 UNIX 类分时系统中，保证多个进程公平地使用系统资源，提供较好的响应时间是调度的主要目标；而在强实时操作系统中，调度器将优先调度具有较高优先级的任务执行。不同的需求将产生不同的调度目标，一个好的调度算法将实现多个目标间的平衡。常见的调度目标有以下几种。

- 公平：确保每个进程获得合理的 CPU 份额。
- 效率：使 CPU 总是处于有效的忙碌中。
- 响应时间：使交互用户的等待时间尽可能短。
- 周转时间：使用户批处理操作等待输出的时间尽可能短。
- 时限保证：保证实时任务在其时限到来前执行完成。
- 优先级：保证高优先级的任务率先得到执行。
- 系统吞吐量：使系统在单位时间内完成尽可能多的任务。
- 能耗：以尽可能低的能耗完成尽可能多的任务。

显然，这些目标有些是相互冲突的，比如在考虑优先级的时候，就不可能兼顾到公平；提高系统的响应时间，就会降低 CPU 的效率，这就需要实现两者间的平衡。例如，在嵌入式实时系统中，常常采用优先级和时间片轮转相结合的调度方式，即不同优先级任务之间按较高优先级优先的原则调度，相同优先级之间则可以按时间片轮转。在嵌入式实时系统中，考虑最多的调度目标是公平、效率、优先级、时限保证等。

在 Linux 中，调度器依据 task_struct 中的 4 个数据项来进行进程调度。这 4 项分别是：policy、nice、counter、rt_priority。policy 代表进程的调度策略；nice 代表进程的动态优先级；counter 代表进程当前剩余的时间片；rt_priority 代表任务实时优先级。

目前，Linux 提供了 3 种调度策略，它们是（使用 task_struct 结构中的 policy 数据项表示）：

- SCHED-OTHER（分时调度策略）：在 Linux 系统中，默认使用该调度策略。

- SCHED-FIFO（实时调度策略）：采用先到先服务的思想。
- SCHED-RR（实时调度策略）：采用轮转的思想。

具体来说，系统首先调度实时进程，当系统中没有实时进程时，再调度分时进程。对于实时进程，系统根据进程的实时优先级决定当前运行进程，系统每次总是选择具有最高优先级的实时进程。SCHED_RR 策略和 SCHED_FIFO 策略的不同之处在于，当采用 SCHED_RR 策略的进程的时间片用完时，系统会给它重新分配时间片，并把它置于就绪队列尾。对于非实时进程，系统总是选择剩余时间片最多的进程投入运行，即选择已占用 CPU 资源最少的进程。当就绪队列中无实时进程，且所有非实时进程的时间片均为 0 时，调度程序会对系统中所有的非实时进程重新分配时间片。这 3 种调度策略的定义在 linux\linux2.4.x\include\linux\sched.h 中，下面是对应的代码：

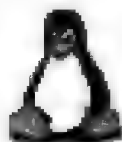
```
/*
 * Scheduling policies
 */
#define SCHED_OTHER      0
#define SCHED_FIFO       1
#define SCHED_RR         2

/*
 * This is an additional bit set when we want to
 * yield the CPU for one re-schedule..
 */
#define SCHED_YIELD            0x10
```

SCHED_OTHER 分时调度策略，是 Linux 系统中使用最多的一种任务调度策略，也是系统默认的任务调度策略。而 SCHED_FIFO、SCHED_RR 调度策略为时间关键的特殊应用程序提供一种精确地控制调度器行为的方法，以便控制调度器如何选择当前的某个就绪程序投入运行。SCHED_OTHER 使用动态优先级策略，其静态优先级必须设为 0。而使用 SCHED_FIFO 和 SCHED_RR 调度策略的任务，使用静态优先级调度算法，优先级可以在 1~99 范围内（值得注意的是，这个范围只是在当前 Linux 版本上有效）。因此，遵循 POSIX 的要求，Linux 还提供了两个系统调用：sched_get_priority_max 和 sched_get_priority_min，用于查询系统有效优先级的最小值和最大值。这两个系统调用的功能是：返回不同策略可使用的优先级范围。其代码在 linux\linux2.4.x\kernel\sched.c 中，下面是对应的代码：

```
asmlinkage long sys_sched_get_priority_max(int policy)
{
        int ret = -EINVAL;

        switch (policy) {
```




```
        case SCHED_FIFO:
        case SCHED_RR:
            ret = 99;
            break;
        case SCHED_OTHER:
            ret = 0;
            break;
        }
        return ret;
}
asmlinkage long sys_sched_get_priority_min(int policy)
{
        int ret = -EINVAL;

        switch (policy) {
        case SCHED_FIFO:
        case SCHED_RR:
            ret = 1;
            break;
        case SCHED_OTHER:
            ret = 0;
        }
        return ret;
}
```

SCHED_FIFO 为先进先出优先级调度，SCHED_RR 为时间片轮转调度，它们都只用于实时任务，即静态优先级大于 0 的情况。这两种策略，按照可抢占优先级调度算法进行。一旦有 SCHED_FIFO 或 SCHED_RR 的任务就绪，将立即抢占 SCHED_OTHER 的任务而投入运行。使用 SCHED_FIFO 的任务将一直运行，直到有更高优先级的任务到达。在任务的执行过程中，任务可以主动调用 Sched_Yield()函数让出处理器，或是调用 exit()函数退出运行。另外，如果任务因申请资源或执行 I/O 操作而被阻塞，也将让出处理器。SCHED_RR 策略可以看作是 SCHED_FIFO 策略的增强，上面关于 SCHED_FIFO 的描述同样适用于 SCHED_RR，除了任务只允许联系运行一个时间片以外，当一个 SCHED_RR 的任务连续运行的时间大于等于时间片的大小时，该任务将被剥夺运行，其 PCB 将放到该任务所处优先级队列的队尾。

在定义 3 种调度策略的同时，Linux 还定义了 SCHED_YIELD 宏，SCHED_YIELD 并不是一种真实调度策略，它只对使用 SCHED_OTHER 调度策略的分时任务有效。当分时任务因为某种原因不需要继续运行时，可以调用 Sched_Yield()函数让出处理器。而调用该函数的任务，其调度策略将被置为 SCHED_YIELD，并将其 PCB 放到该任务所处优先级队列的队尾。

nice、counter 分别保存了进程的优先级和剩余时间片（以时钟滴答计算）。在 UNIX 系统中，nice 值将影响一个进程获取的处理器时间。nice 取值范围为-20~+19，并且 nice 值越小，表示任务的优先级越高，-20 表示最高的优先级，+19 表示最低优先级。在 Linux 2.4.x 中，nice 直接换算为任务的滴答数，其相关代码在 linux\linux2.4.x\kernel\sched.c 中：

```
* We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x)  ((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x)  ((x) >> 1)
#elif HZ < 800
#define TICK_SCALE(x)  (x)
#elif HZ < 1600
#define TICK_SCALE(x)  ((x) << 1)
#else
#define TICK_SCALE(x)  ((x) << 2)
#endif

#define NICE_TO_TICKS(nice) (TICK_SCALE(20-(nice))+1)
```

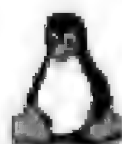
为了将系统时间片大小维持在 50 毫秒左右，Linux 将根据系统的时钟频率对一个 nice 对应的滴答数进行调整。调整的方法是频率越低，tick/nice 越小。具体比例，读者可以阅读上面的代码。

进程创建时，nice 继承自父进程，而 counter 值为父进程的一半，同时将父进程的 counter 也减小为原来的一半。这样做的目的是在父子进程之间共享动态优先级，以保持整个系统中的动态优先级不变，使得调度算法对所有任务来说更加公平。当然，这个过程只在进程刚创建时有效，即第一个时间片，进程的长期行为还是受 nice 的控制。下面是 do_fork()函数中处理 nice 和 counter 的部分代码：

```
p->counter = (current->counter + 1) >> 1;//子进程 counter 的时间片置为父进程的一半
current->counter >>= 1;                  //父进程 counter 的时间片也减小为原来的一半
if (!current->counter)                   //检查是否需要重新调度，如果是，则设置标志位
        current->need_resched = 1;
```

3.3.2 调度方法分类

虽然任务调度的主要目的都是合理地分配处理器资源，但在不同的操作系统中所采用



的调度算法却不尽相同。自20世纪70年代以来，实时调度算法研究非常活跃，在这一领域中存在大量的研究成果，但概括起来，常用的实时调度算法可分为优先级调度和轮转调度两种。其中，优先级调度对于实时操作系统必不可少。

调度算法有多种分类方式，按照调度决策过程是否在运行时进行，调度算法可以分为预先计算的（离线）和动态计算的（在线）。离线调度在任务时间运行前进行，包括确定每个周期任务的具体运行时间，以及间发任务和非周期任务将占用的时间片。在线调度在任务到达时对任务进行调度。用于在线调度的算法必须快速、高效。如果在线调度算法占用过多的系统时间，将使实时任务无法获得足够的时间，在时限到来前完成规定的操作，这样的算法是毫无用处的。

按照优先级在任务生命周期中是否可以改变，优先级调度可以分为静态优先级调度和动态优先级调度。在静态优先级调度算法中，任务的优先级可以根据任务的某些固定不变的属性来确定，比如任务的周期、任务的绝对时限、任务的执行时间、任务的松弛时间等；任务的优先级也可以由用户手工指定，在创建任务时，将任务的优先级作为一个参数传递给系统 API。在动态优先级调度算法中，任务的优先级可以由任务的某些动态属性确定，如任务的相对时限。在任务的优先级由用户手工指定的情况下，任务的优先级也可以通过调用系统 API 在运行中更改。

按照正在执行的任务可否被抢占，优先级调度分为可抢占调度（preemptive scheduling）和不可抢占调度（non-preemptive scheduling）；而按照操作系统内核是否可被抢占，可抢占调度又可分为内核可抢占调度和内核不可抢占调度。

- 内核可抢占调度

上面提到的可抢占调度是指一般意义下的低优先级任务被高优先级任务抢占，在强实时嵌入式操作系统中，内核是否可被抢占也是衡量调度算法实时性的一个重要指标。所谓内核可被抢占，是指在低优先级任务调用系统 API 而陷入系统内核时，如果发生中断，并且在中断处理过程中如有更高优先级的任务就绪，则在退出中断服务程序的时候，进行任务重调度，使得高优先级的任务可以立即开始运行，而不必返回到低优先级任务中，等待系统 API 执行完成。图3-3所示为内核可抢占调度的一个演示示例。

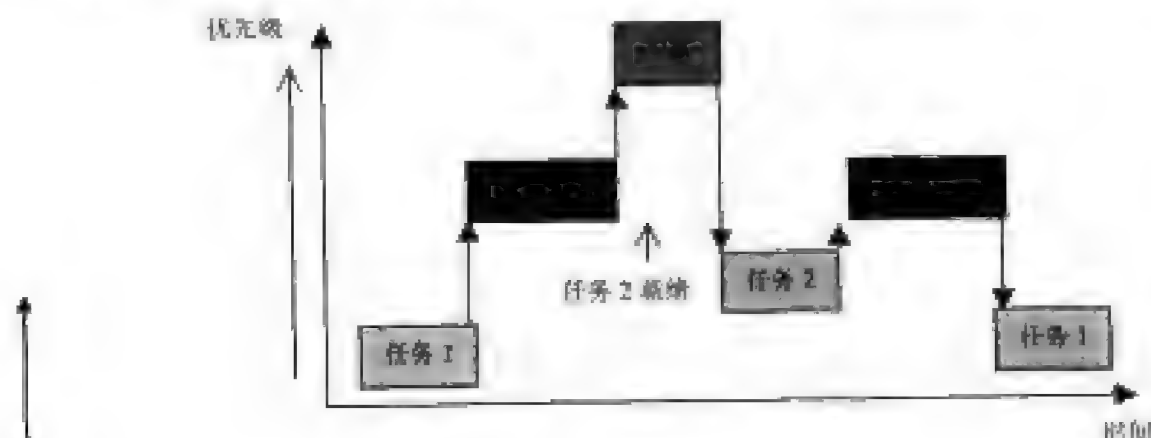


图3-3 内核可抢占调度的演示示例

- 内核不可抢占调度

内核不可抢占调度是指在系统中，如果用户通过调用系统 API 进入内核执行后，即使有高优先级的任务就绪，也必须等待该系统 API 执行完成后，才能被调度运行。即内核代码的执行不能被任务中断。图 3-4 所示为内核不可抢占调度的一个演示示例。

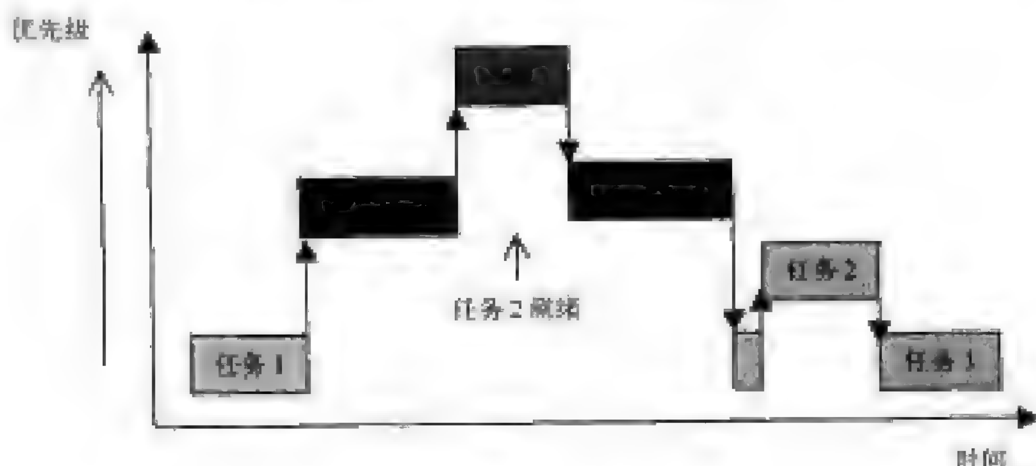


图 3-4　内核不可抢占调度的示例

优先级算法规定了不同优先级的任务之间的执行顺序，对于相同优先级的任务而言，有两种调度策略可供使用：

- 先来先服务（FIFO）。是指对于同优先级的任务，谁先开始运行则谁将一直占用处理器，直到有更高优先级的任务到达，或是任务运行完成。
- 同优先级时间片轮转调度。为了使实时系统中优先级相同的任务具有平等的运行权利，当有两个或多个就绪任务具有相同的优先级且它们是就绪任务中优先级最高的任务时，任务调度程序将按照这组任务就绪的先后次序调度第一个任务，让它运行一段时间，运行的这段时间称为时间片（time slice）。当任务运行完一个时间片后，该任务即使还没有运行完成，也必须释放处理器让下一个与它具有相同优先级的任务运行（假设这时没有更高优先级的任务就绪）。该任务将被排到同优先级就绪任务链的链尾，等待下一个时间片到来再次运行。

采用时间片轮转调度算法时，任务的时间片大小要选择适当。时间片大小的选择会影响系统的性能和效率。时间片太大，时间片轮转调度就没有意义；时间片太小，任务切换过于频繁，处理器开销将增大，真正用于运行应用程序的时间将会减小。

在具体的系统中，所有任务的时间片可以采用一个固定的大小，也可以不同任务有不同的时间片大小。

同优先级轮转调度常作为优先级调度的补充使用，图 3-5 所示是一个使用了同优先级轮转的优先级调度的示意图。



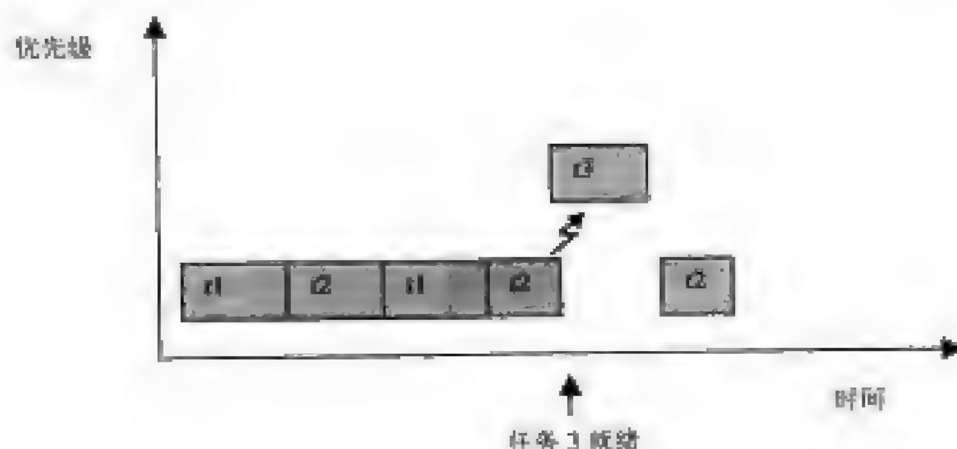


图 3-5　可抢占调度和时间片轮转调度并存的一个实例

3.3.3　经典常用实时调度算法

实时系统中，经典的常用调度算法包括用户指定优先级调度、速率单调调度（Rate Monotonic）、最早时限优先调度、最少松弛时间调度等。

1. 用户指定优先级调度

用户指定优先级调度是嵌入式实时系统中使用最广泛的调度算法。在此种调度下，任务的优先级由用户根据需要手工指定，通常是作为系统调用中创建任务 API 的一个参数传递给系统。一般情况下，系统都会提供在运行中改变任务优先级的调用。这种调度算法，把确定任务优先级的责任完全交给用户承担。任务的优先级与任务的其他属性无关。相对其他调度算法来说，用户指定优先级调度最为灵活，对任务集的限制最小，适应范围广。但是，任务优先级需要用户手工预先计算，从而增大了用户的工作量，并且该算法不能保证低优先级的任务能够得到运行。但是由于这种算法灵活，适应范围广，实现简单，大多数嵌入式实时系统中都提供了该种调度算法。

2. 速率单调调度（Rate Monotonic）算法

速率单调调度又可简称为 RM 调度算法，是研究和应用最为广泛的实时调度算法。RM 调度算法是单处理器静态优先级可抢占调度算法。为了简化算法的分析过程，RM 对调度环境作以下假设：

（1）任务总可以被抢占，抢占的代价可以忽略不计。

（2）只考虑任务的处理器需求，内存、I/O 和其他资源请求忽略不计。

（3）任务间相互独立，不存在先后关系。

（4）任务集中所有任务都是周期任务。

（5）任务的相对时限等于周期。

上述假设极大地简化了 RM 算法的分析。假设（1）使调度算法能够在任何时刻抢占任何任务，并在任何时候恢复被抢占的任务，而且没有任何代价。因此，任务被抢占的时间的数量不会增加系统的负载。根据假设（2），调度可行性只需通过判定是否存在足够的处理器时间，而不用考虑内存或其他限制条件。假设（3）使任务间不存在先后关系，任务的释放时间不依赖于其他任务的结束时间。假设（5）保证任何时刻任何任务只可能存在一个实例。

在 RM 调度算法中，任务的优先级排列顺序与任务周期的排列相反，如果任务 1 的周期小于任务 2，则任务 1 的优先级高于任务 2，高优先级任务可以抢占低优先级任务执行。以下是一个 RM 调度的应用实例。

假设存在如下任务集 S，如表 3-1 所示。

表 3-1　3 个周期任务的任务集 S

任　　务	执行时间 C	周期 P
1	20	100
2	30	150
3	70	210

其中，记任务 i 的优先级为 PR_i，因为 $P_1<P_2<P_3$ 所以 $PR_1>PR_2>PR_3$。

任务的运行情况如图 3-6 所示。

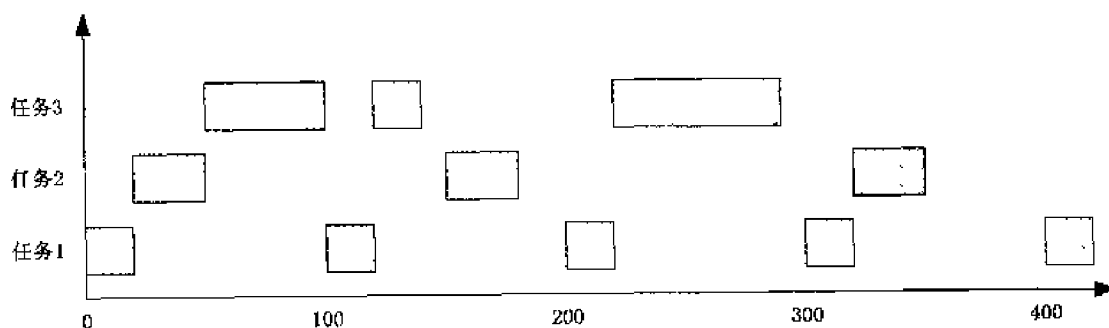


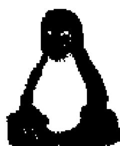
图 3-6　3 个任务的 RM 调度示意图

前面已经介绍了 RM 调度对任务集的约束条件和任务优先级确定方法，但是如何确定一个任务集可以被 RM 调度算法调度呢？为了说明这个问题，首先介绍一些相关概念。

任务资源利用率指任务占用处理器时间的大小，对于周期任务，可以使用执行时间和周期的比值来衡量。记任务 T_i 的执行时间为 C_i，周期为 P_i，时限为 D_i，则任务 i 的资源利用率为 C_i/P_i。

如果一个任务集可以被 RM 调度算法调度，并且延长任意任务的执行时间将导致该任务集不再能被调度，则称该任务集完全利用处理器。值得注意的是，完全利用处理器并不意味着在$[0,\infty]$间，处理器一直处于工作状态。

最小处理器利用率下限指所有完全利用处理器的任务集的总资源利用率的最小值。



关于 RM 调度的可调度条件，有如下定理：

定理 1 如果任务集的总资源利用率不大于 $n(2^{1/n}-1)$，RM 将调度所有任务满足其时限要求，其中 n 是被调度的任务的数目。即对于任务集 $S\{T_1, T_2, \ldots, T_n\}$，如果满足

$$\frac{C_1}{P_1}+\frac{C_2}{P_2}+\cdots+\frac{C_n}{P_n}\leqslant n\left(2^{1/n}-1\right)$$

则该任务集可被 RM 调度。

值得注意的是，上述条件是充分非必要条件，当总资源利用率高于 $n(2^{1/n}-1)$时，实时任务集仍旧有可能被 RM 算法调度。

仍然以表 3-1 的例子来说明 RM 可调度条件的使用方法。

假设任务集中有 3 个任务，可记为 T_1、T_2、T_3，其资源利用率分别为 $\frac{20}{100}$、$\frac{30}{150}$、$\frac{70}{210}$，所以任务集的总资源利用率为 $\frac{20}{100}+\frac{30}{150}+\frac{70}{210}\approx 0.2+0.2+0.33333=0.73333$，而 3 个任务的 RM 调度算法的最小处理器利用率下限是 $3\left(2^{1/3}-1\right)=0.77976$，因为 $0.733333<0.77976$，所以根据定理 1，可以知道该任务集可以由 RM 调度算法调度。

下面来看一下，当不满足定理 1 时，仍然可以被调度的情况。将任务集 S 中任务 3 的周期改为 80 时，其他不变，记为 S'。任务集的总资源利用率为 $\frac{20}{100}+\frac{30}{150}+\frac{80}{210}\approx 0.2+0.2+0.38095=0.78095>0.77976$，因此，由定理 1 无法判断 S'是否可由 RM 算法调度。通过画出 S'的调度图（如图 3-7 所示），可以看出该任务集实际上可由 RM 算法调度。

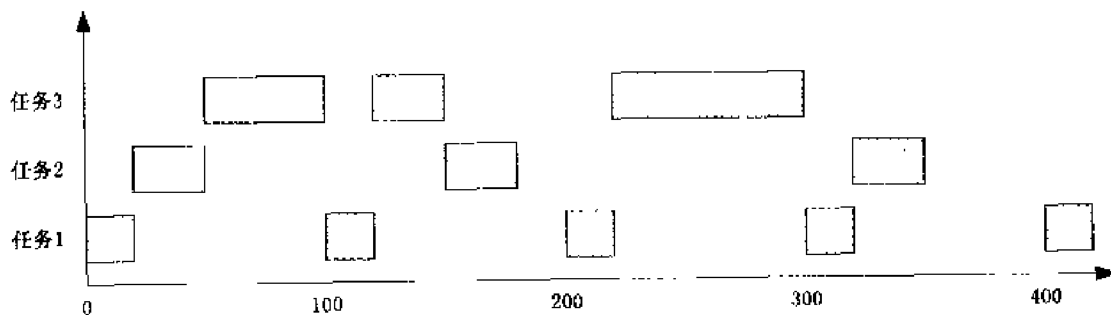


图 3-7 3 个任务的 RM 调度示意图

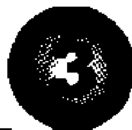
因此，定理 1 只是 RM 调度判定的充分条件。接下来，讨论 RM 调度算法的充要条件。首先，定义相关符号表示如下：

$$W_i(t)=\sum_{j=1}^{i}C_j\left\lceil t/P_j\right\rceil$$

$$L_i(t)=W_i(t)/t$$

$$L_i(t)=\min_{0<t\leq P_i}L_i(t)$$

$$L=\max\left(L_i\right)$$



其中，$W_i(t)$是 T_1、T_2、…、T_n 在[0,t]间释放的所有实例的处理器时间需求。如果所有任务均在 0 时刻释放，基于 RM 调度，将 T_i 在 t'时结束，则 $W_i(t')=t'$。

RM 调度的充要条件可表达如下：

由 n 个周期任务组成的任务集（$P_1 \leqslant P_2 \leqslant \cdots \leqslant P_n$），如果 $L_i \leqslant 1$，L_i 可调度。

通过计算公式 $W_i(t)$，将使判断 $W_i(t)$是否小于 t 变得十分简单。$W_i(t)$通常保持不变，仅在任务释放点处发生变化。因此，只需在以下时刻计算 $W_i(t)$：

$$\tau_i = \left\{ lP_j \middle| j = 1,\cdots,i; l = 1,\cdots,\lfloor P_i / P_j \rfloor \right\}$$

关于 RM 算法的可调度条件，存在以下两个定理：

定理 2　任务 T_i 可由 RM 调度，如果

$$\min_{t \in \tau_i} W_i(t) \leqslant t$$

定理 3　整个任务集 T 均可由 RM 调度，如果

$$\max_{i \in (1,2,\cdots,n)} \{\min_{t \in \tau} W_i(t)/t\} \leqslant 1$$

接下来，看一个使用定理 3 的应用实例。在任务集 S'中加入一个新任务，周期为 400，执行时间为 100。构成的新任务集 S''如表 3-2 所示。

表 3-2　任务集 S''

i	e_i	P_i
1	20	100
2	30	150
3	80	210
4	100	400

则：

$$\tau_1 = \{100\}$$
$$\tau_2 = \{100, 150\}$$
$$\tau_3 = \{100, 150, 200, 210\}$$
$$\tau_4 = \{100, 150, 200, 210, 300, 400\}$$

以下分析各任务的可调度性。如果任务 T_i 的 $W_i(t)$总低于 $W_i(t)=t$，则 T_i 可调度。可调度条件的数学表达如下所示：

如果 $e_1 \leqslant 100$，则任务 T_1 可调度；

如果 $e_1 + e_2 \leqslant 100$ 或

$2e_1 + e_2 \leqslant 150$，则任务 T_2 可调度；

如果 $e_1 + e_2 + e_3 \leqslant 100$

或 $2e_1 + e_2 + e_3 \leqslant 150$

或 $2e_1 + 2e_2 + e_3 \leqslant 200$

或 $3e_1 + 2e_2 + e_3 \leqslant 210$，则任务 T_3 可调度；

如果 $e_1 + e_2 + e_3 + e_4 \leqslant 100$

或 $2e_1+e_2+e_3+e_4\leqslant 150$

或 $2e_1+2e_2+e_3+e_4\leqslant 200$

或 $3e_1+2e_2+e_3+e_4\leqslant 210$

或 $3e_1+2e_2+2e_3+e_4\leqslant 300$

或 $4e_1+3e_2+2e_3+e_4\leqslant 400$，则任务 T_4 可调度。

从上述表达式可以看出，任务 T_1、T_2 和 T_3 可调度，T_4 不可调度。

定理 4 RM 调度是最优静态优先级算法。

也就是说，对某个任务集，如果任意一个静态优先级调度算法能够调度该任务集，则 RM 调度算法也能调度该任务集。

正是由于 RM 是最优的静态优先级算法，因此，RM 算法在学术研究和实际工程应用中得到了广泛的关注。

上面，介绍了 RM 调度算法的基本情况。但是在上面的介绍中，只考虑了符合假设的理想情况。在实际的工程应用中，还会遇到很多实际的应用，不能满足其假设条件，例如，任务不是周期任务和任务时限不等于周期等。放宽对任务的限制条件，是目前实时调度算法研究的热点问题。目前该领域已有大量的研究成果，感兴趣的读者可以参考 C.M.Krishna 的经典教材《Real-time Systems》。

3. 最早时限优先调度（Early Deadline First）

最早时限优先调度算法也可简称为 EDF 算法。根据 EDF 算法，处理器将分配给当前距离绝对时限最近的任务。EDF 算法是动态调度算法，任务的优先级不是固定的，而是根据各任务距离其绝对时限的接近程度决定。

在讨论 EDF 算法的过程中，除了任务可以是非周期任务外，将沿用 RM 算法的假设。即：

- 任务总可以被抢占，抢占的代价可以忽略不计。
- 只考虑任务的处理器需求，内存、I/O 和其他资源请求忽略不计。
- 任务间相互独立，不存在先后关系。

设存在表 3-3 所示任务集 S。

表 3-3 任务集 S

任　务	释放时间	执行时间	绝对时限
T_1	0	10	30
T_2	4	3	10
T_3	5	10	25

当 T_1 释放时，系统中只存在一个任务等待运行，因此 T_1 立刻开始运行；T_2 在时刻 4 释放，且 $d_2<d_1$，因此 T_2 的优先级高于 T_1，T_2 抢占 T_1；T_3 在时刻 5 释放，且 $d_3>d_2$，因此 T_3 的优先级低于 T_2，T_2 继续运行。当 T_2 在时刻 7 完成，T_3 开始运行，因为 T_3 的优先级高于 T_1。T_3 在时刻 15 结束运行，T_1 继续运行。整个调度过程如图 3-8 所示。

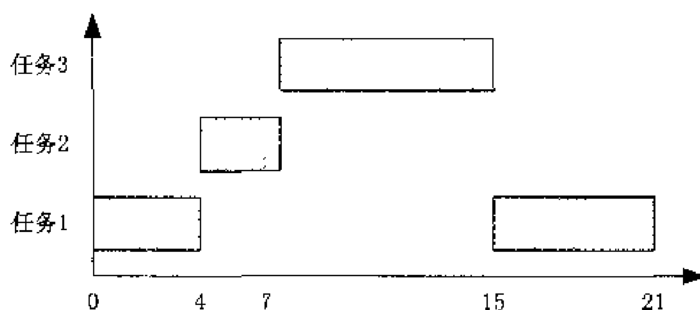


图 3-8　3 个任务的 EDF 示意图

关于某任务集是否可由 EDF 算法调度，有以下两个定理。

定理 5　EDF 算法是单处理器最优调度算法。也就是说，如果 EDF 算法不能调度某任务集，则其他算法也不能调度该任务集。

定理 6　如果任务均为周期任务，且相对时限等于周期，并且任务集的总资源利用率不大于 1，则任务集可由 EDF 算法调度。

假设任务的周期等于任务的相对时限，则表 3-4 所示的任务集 S'的总资源利用率为：

$$\frac{10}{30}+\frac{3}{10}+\frac{10}{25}\approx 0.33333+0.3+0.4=0.73333<1$$

表 3-4　7 任务集 *S'*

任　　务	释 放 时 间	执 行 时 间	周　　期
T_1	0	10	30
T_2	4	3	10
T_3	5	10	25

由定理 6，可以知道，该任务集可以被 EDF 算法调度。这与前面在 *S* 任务集中的分析是吻合的。

对于相对时限不等于周期的情况，不存在简单的可调度判定方法。在这种情况下，需要通过一个模拟调度过程，确定任务集中任务的时限是否满足，其基本方法如下：

定义 $U=\sum_{i=1}^{n}(e_i/P_i)$，$d_{\max}=\max_{1\le i\le n}\{d_i\}$，$P=lcm(P_1,\cdots,P_n)$。设 $h_T(t)$是任务集 T 中绝对时限小于 t 的所有任务的总执行时间。则当 $u>1$，或存在

$$t<\min\left\{P+d_{\max},\frac{u}{1-u}\max_{1\le i\le n}\{P_i-d_i\}\right\}$$

导致 $h_T(t)>t$ 时，任务集不能被 EDF 算法调度。

EDF 算法同样是实时调度中广泛应用的算法。而且它也同样存在假设条件太苛刻的问题。放宽对任务集的假设条件，是实时调度算法的重要发展方向。



3.3.4 多处理器调度算法

讨论调度算法时，都是在单处理器的情况下进行的。随着嵌入式实时应用功能日益复杂，实时应用对实时系统性能的要求越来越高，并且实时计算的工作模式开始从简单的单节点集中处理，转向分布式的协同计算。现代嵌入式实时操作系统应该能够支持同构、异构的多处理器系统，以满足复杂、很高性能要求的实时系统的需要。因此，研究高效、可靠、实用的实时多处理器调度算法是当前实时系统研究的重要问题。

绝大多数超过两个处理器的任务分配和调度问题是 NP 问题。因此，启发式算法是必不可少的。大多数启发式算法是以单处理器调度（任务集在一个处理器上运行）为基础的，因为单处理器调度易于实现。

多处理器调度算法的设计可以分为两步：首先将任务分配到不同的处理器，然后在每个处理器上运行单处理器调度。如果一个或多个调度不可行，则重新分配任务。图 3-9 所示是上述方法的演示，该方法存在多个变种，例如，可以在每个任务分配后检查可调度性。

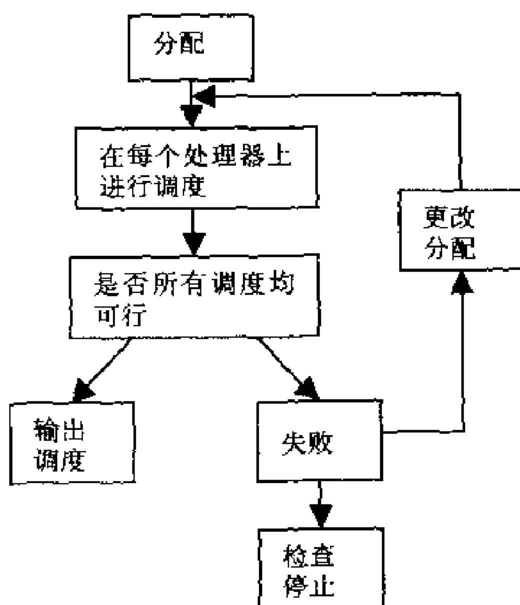


图 3-9 多处理器启发式算法流程

3.3.5 Linux 进程调度

1. Linux 的调度算法

在本章前面几个小节中，已经分析了 Linux 的调度策略和调度依据。接下来，进一步分析 Linux 的调度算法。如前所述，Linux 支持分时调度策略（SCHED_OTHER）和实时调度策略（SCHED_FIFO、SCHED_RR）。接下来，介绍分时调度算法的实现方法。

Linux的分时调度是一种基于时间片的动态优先级调度。任务新创建时，被赋予一个优先级（继承自父进程），该优先级用task_strcut（即Linux中任务的PCB）的nice数据项表示。PCB中的counter则表示进程当前剩下的时间片，在系统运行过程中，nice值不变，counter不断递减。counter递减的过程是在系统时钟的中断处理例程中，调用update_process_times()函数，对当前正在运行的进程的counter值减1。当所有任务的counter都递减为0后，系统将对所有的任务重新分配时间片。下面是update_process_times()函数的实现代码（在linux\linux2.4.x\kernel\timer.c中）：

```
void update_process_times(int user_tick)
{
	struct task_struct *p = current;
	int cpu = smp_processor_id(), system = user_tick ^ 1;

	update_one_process(p, user_tick, system, cpu);
	if (p->pid) {                              /*判断当前进程是否是0号进程*/
		if (--p->counter <= 0) {             /*counter值递减*/
			p->counter = 0;
			/*
			 * SCHED_FIFO is priority preemption, so this is
			 * not the place to decide whether to reschedule a
			 * SCHED_FIFO task or not - Bhavesh Davda
			 */
			if (p->policy != SCHED_FIFO) {
				p->need_resched = 1;       /*设置进程重调度标志为1*/
			}
		}
		if (p->nice > 0)
			kstat.per_cpu_nice[cpu] += user_tick;
		else
			kstat.per_cpu_user[cpu] += user_tick;
		kstat.per_cpu_system[cpu] += system;
	} else if (local_bh_count(cpu) || local_irq_count(cpu) > 1)
		kstat.per_cpu_system[cpu] += system;
}
```

输入参数user_tick为1表示当前tick用于用户操作，为0表示当前tick用于系统操作。这样，Linux就可以统计一个进程的用户时间和系统时间。update_process_times()函数中首先判断当前进程是否为0号进程，如果不是，则将进程counter值递减；如果counter小于0，则将其重置为0。接下来的代码，对SCHED_RR和SCHED_OTHER任务进行处理，因为当前任务时间片已经为0，所以将进程重调度标志置为1。函数的最后部分，则是对任务



的用户时间 per_cpu_user 和系统时间 per_cpu_system 进行的统计。

在系统运行过程中，counter 作为任务的动态优先级使用，因为在调度器选择下一个就绪过程的时候，对分时任务，将部分使用 counter 作为进程的权值。由于 counter 值在运行过程中是不断变化的，因此，说 Linux 的分时调度是一种动态优先级调度。

Linux 系统中，使用 goodness()函数计算任务的权值，调度器将选择权值最大的任务投入运行。下面是 Goodness()函数的实现代码（在 Linux2.4.0\kernel\Sched.c 中）：

```
static inline int goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)
{
        int weight;

        /*
         * select the current process after every other
         * runnable process, but before the idle thread.
         * Also, dont trigger a counter recalculation.
         */
        weight = -1;
        /*判断如果任务的调度策略被置为 SCHED_YIELD 的话，则置权值为-1，返回*/
        if (p->policy & SCHED_YIELD)
                goto out;

        /*
        * Non-RT process - normal case first
        先处理分时任务的情况，因为 Linux 中绝大多数任务都使用分时调度策略，这样
做可以提高系统的整体效率
         */
        if (p->policy == SCHED_OTHER) {
                /*
                 * Give the process a first-approximation goodness value
                 * according to the number of clock-ticks it has left.
                 *
                 * Don't do any other calculations if the time slice is
                 * over..
                 */
                weight = p->counter;/* 将进程的权值置为进程的 counter 值*/
        /*
        如果当前进程的 counter 为 0，则表示当前进程的时间片已经用完了，不再对权值
进行进一步的计算，直接返回
         */
                if (!weight)
                        goto out;
```

```
#ifdef CONFIG_SMP
		/* Give a largish advantage to the same processor... ... */
		/* (this is equivalent to penalizing other processors) */
		if (p->processor == this_cpu)
			weight += PROC_CHANGE_PENALTY;
#endif

	/*
对进程权值进行微调，如果进程的内存空间使用当前正在运行的进程的内存空间，则权
值加 1
	*/
		if (p->mm == this_mm || !p->mm)
			weight += 1;
	/*
将权值加上 20 与进程优先级 nice 的差，实际上，分时进程的权值主要是由 counter 值和
nice 值组成
*/
		weight += 20 - p->nice;
		goto out;
	}

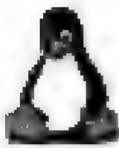
	/*
对实时进程进行处理，实时进程的权值等于其静态优先级加上 1000 的偏移。加上 1000，
可以确保实时进程的优先级远大于分时进程的优先级
	*/
	weight = 1000 + p->rt_priority;
out:
	return weight;
}
```

由此，可以看出，通过一个简单的 goodness()函数，Linux 实现了多种调度策略的统一处理，设计思想可以说是相当巧妙。

另外，需要补充的一点是，Linux 支持传统的对称多处理器系统，但是在嵌入式环境中使用很少，本书不涉及这方面的内容，感兴趣的读者，可以参考由 Curt Schimmel 所著的《UNIX Systems for Modern Architectures－Symmertric Multiprocessing and Caching for Kernel Programmers》和由毛德操、胡希明老师所著的《Linux 内核源代码情景分析》。

2. 任务的基本操作

为了能控制任务运行过程，操作系统都会提供相应的任务控制系统调用。一般来说，任务控制包括创建、销毁、挂起、唤醒等。下面来看一下 Linux 任务管理功能的具体实现



方法。

（1）进程号。

为了实现任务的管理，必须有能惟一标识任务的方法。在 Linix 系统中，使用进程号来惟一标识一个进程。进程号是非负值，在大多数平台上都定义为一个 int 类型。Linux 系统中提供了一个静态函数 get_pid()，专门负责进程号的生成，如果申请分配进程号成功，该函数返回新的进程号，否则返回 0。因为 0 进程号已经在 Linux 运行过程中将一直被 init 进程占用。下面是该函数代码（在 linux\linux2.4.x\kernel\fork.c 中）：

```
static int get_pid(unsigned long flags)
{
	static int next_safe = PID_MAX;
	struct task_struct *p;
	int pid, beginpid;

	if (flags & CLONE_PID)
		return current->pid;

	spin_lock(&lastpid_lock);
	beginpid = last_pid;
	if((++last_pid) & 0xffff8000) {
		last_pid = 300;		/* Skip daemons etc. */
		goto inside;
	}
	if(last_pid >= next_safe) {
inside:
		next_safe = PID_MAX;
		read_lock(&tasklist_lock);
	repeat:
		for_each_task(p) {
			if(p->pid == last_pid ||
			   p->pgrp == last_pid ||
			   p->tgid == last_pid ||
			   p->session == last_pid) {
				if(++last_pid >= next_safe) {
					if(last_pid & 0xffff8000)
						last_pid = 300;
					next_safe = PID_MAX;
				}
				if(unlikely(last_pid == beginpid))
					goto nomorepids;
				goto repeat;
```

```
			}
			if(p->pid > last_pid && next_safe > p->pid)
				next_safe = p->pid;
			if(p->pgrp > last_pid && next_safe > p->pgrp)
				next_safe = p->pgrp;
			if(p->tgid > last_pid && next_safe > p->tgid)
				next_safe = p->tgid;
			if(p->session > last_pid && next_safe > p->session)
				next_safe = p->session;
		}
		read_unlock(&tasklist_lock);
	}
	pid = last_pid;
	spin_unlock(&lastpid_lock);

	return pid;

nomorepids:
	read_unlock(&tasklist_lock);
	spin_unlock(&lastpid_lock);
	return 0;
}
```

（2）进程控制块。

Linux 中的进程控制块使用 task_struct 数据结构来表示。task_struct 存储了一个进程的所有信息。Linux 中对任务的管理都要用到该数据结构。

其源代码定义如下（在文件 linux\linux2.4.x\include\linux\sched.h 中）：

```
struct task_struct {
	/*
	 * offsets of these are hardcoded elsewhere - touch with care
	 */
	volatile long state;	/*进程状态 -1 不可运行， 0 可运行， >0 停止状态*/
	unsigned long flags;	/*进程标志*/
	int sigpending;		/*进程待决信号标志*/
	mm_segment_t addr_limit;	/*线程虚拟地址空间的上限：
					0-0xBFFFFFFF  用户线程
					0-0xFFFFFFFF  内核线程
				*/
	struct exec_domain *exec_domain;	/*执行域，标识应用程序版本之间的差异*/
	volatile long need_resched;		/*是否需要重调度*/
```




```
        unsigned long ptrace;                         /*能否跟踪标志*/

        int lock_depth;                               /*锁定深度*/

/*
 * offset 32 begins here on 32-bit platforms. We keep
 * all fields in a single cacheline that are needed for
 * the goodness() loop in schedule().
 */
        long counter;              /*进程当前还拥有的时间片*/
        long nice;                 /*进程的 nice 值，来自 UNIX 系统，相当于进程优先级*/
        unsigned long policy;      /*进程调度策略*/
        struct mm_struct *mm;      /*进程的内存空间结构*/
        int processor;             /*处理器号*/
        /*
         * cpus_runnable is ~0 if the process is not running on any
         * CPU. It's (1 << cpu) if it's running on a CPU. This mask
         * is updated under the runqueue lock.
         *
         * To determine whether a process might run on a CPU, this
         * mask is AND-ed with cpus_allowed.
         */
        unsigned long cpus_runnable, cpus_allowed;
        /*
         * (only the 'next' pointer fits into the cacheline, but
         * that's just fine.)
         */
        struct list_head run_list;                      /*run（就绪）进程链表*/
        unsigned long sleep_time;                       /*睡眠时间*/

        struct task_struct *next_task, *prev_task;      /* 进程链接信息*/
        struct mm_struct *active_mm;
        struct list_head local_pages;
        unsigned int allocation_order, nr_local_pages;

/* task state */
        struct Linux_binfmt *binfmt;/*应用程序的格式，如 a.out、elf、coff 等*/
        int exit_code, exit_signal;
        int pdeath_signal;   /*The signal sent when the parent dies*/
        /* ??? */
        unsigned long personality;/*进程的"个性"，由进程对应的执行程序决定*/
        int did_exec:1;
```

```
        pid_t pid;                              /*进程号*/
        pid_t pgrp;                             /*进程组号*/
        pid_t tty_old_pgrp;
        pid_t session;                          /*会话号*/
        pid_t tgid;
        /* boolean value for session group leader */
        int leader;                             /*会话的领导者*/
        /*
         * pointers to (original) parent process, youngest child, younger sibling,
         * older sibling, respectively.  (p->father can be replaced with
         * p->p_pptr->pid)
         */
        struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;/*进程家族信息，分别
是祖先、双亲、孩子、右兄弟、左兄弟*/
        struct list_head thread_group;

        /* 进程号(PID) hash 表链接信息*/
        struct task_struct *pidhash_next;
        struct task_struct **pidhash_pprev;

        wait_queue_head_t wait_chldexit;        /*用于 wait4 系统调用的标志位*/
        struct completion *vfork_done;          /*用于 vfork 系统调用的标志位*/
        unsigned long rt_priority;              /*进出的实时优先级*/
        unsigned long it_real_value, it_prof_value, it_virt_value;    /*实时、概况、虚拟时钟
的值（到期时间）*/
        unsigned long it_real_incr, it_prof_incr, it_virt_incr;       /*实时、概况、虚拟时钟
的定时间隔*/
        struct timer_list real_timer;
        struct tms times;
        unsigned long start_time;               /*进程创建时间*/
        long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];/*该进程在各处理器上
运行的用户时间和系统时间*/
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific
*/
        unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;/*页面异常统计和
换入换出次数统计信息*/
        int swappable:1;                        /*是否可以交换*/
/* process credentials */
        uid_t uid,euid,suid,fsuid;              /*分别是用户、有效用户、备份用户、文件系统用户
标识号*/
        gid_t gid,egid,sgid,fsgid;              /*分别是组、有效组、备份组、文件系统组标识号*/
        int ngroups;
```




```
    gid_t    groups[NGROUPS];
    kernel_cap_t    cap_effective, cap_inheritable, cap_permitted;/*进程能力标志，规定了进程是否能进行许多特权操作*/
    int keep_capabilities:1;
    struct user_struct *user;        /*用户结构*/
/* limits */
    struct rlimit rlim[RLIM_NLIMITS];  /*一个数组，存储了系统对该进程能使用的各种资源的数量的限制*/
    unsigned short used_math;
    char comm[16];
/* file system info */
    int link_count, total_link_count;     /*文件系统信息链接信息*/
    struct tty_struct *tty; /* tty 信息 */
    unsigned int locks;           /*进程拥有的文件锁数目*/
/* ipc stuff */
    struct sem_undo *semundo;         /*信号量 undo 结构，用于信号量的撤销*/
    struct sem_queue *semsleeping;
/* CPU-specific state of this task */
    struct thread_struct thread;        /* */
/* filesystem information */
    struct fs_struct *fs;           /*进程文件系统信息*/
/* open file information */
    struct files_struct *files;         /*进程打开文件信息*/
/* namespace */
    struct namespace *namespace;        /*进程名字空间结构*/
/* signal handlers */
    spinlock_t sigmask_lock;          /*信号保护和阻塞自旋锁*/
    struct signal_struct *sig;         /*信号结构*/

    sigset_t blocked;             /*进程阻塞标志，可用于信号处理等*/
    struct sigpending pending;         /*待决信号的链表头*/

    unsigned long sas_ss_sp;
    size_t sas_ss_size;
    int (*notifier)(void *priv);
    void *notifier_data;
    sigset_t *notifier_mask;

/*进程组跟踪信息*/
    u32 parent_exec_id;
    u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty */
```

```
        spinlock_t alloc_lock;          /*内存、文件、文件系统、tty 等的分配操作时的保
护标志*/

/* journalling filesystem info */
        void *journal_info;             /*日志文件系统信息*/

#ifdef CONFIG_SYSCALLTIMER
        int curr_syscall;
#endif
};
```

（3）任务创建。

任务的创建可分为以下几步：

1）给任务分配任务控制块。

2）初始化任务的基本信息：进程号、优先级、时间片等基本信息。

3）初始化与任务管理相关的其他信息，比如任务内存空间信息、文件访问信息、任务间通信信息等。

4）初始化任务的链接信息，即将任务加入系统的任务管理数据结构中。比如，将进程加入进程树中。

5）设置任务的运行代码位置。

6）开始任务运行（上下文切换）。

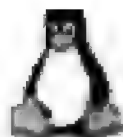
传统实时系统中，上述任务的创建工作一般由一个或两个统一的系统调用完成，该系统调用包含运行代码的位置，在任务创建的初始化工作完成后，任务将直接从运行代码位置处开始执行，调用任务所创建的 API 任务和新创建的任务之间没有直接的联系。

而在 UNIX 类操作系统中，任务的创建方式有所不同。父进程调用 fork 系统调用创建子进程，子进程共享父进程的大部分资源，包括进程的运行代码、文件访问信息、内存信息等。

父子进程都从 fork 系统调用中返回，接着往下执行。所不同的是，在父进程中，fork 调用返回子进程的进程号；而在子进程中，fork 调用返回 0。惟一的例外情况就是 0 号进程是在系统初始化时，由初始化代码手工创建的。下面是一个任务创建实例的代码：

```
# include <stdio.h>

main()
{
        int pid;
        pid=fork();
        if(pid>0)
{
        printf("Father !, the child pid is %d\n",pid);
        exit(0);
```




```
}
if(pid==0)
{
        printf("Child\n");
        exit(0);
}
if(pid<0)
{
        printf("Error,No enough memory!\n");
exit(1);
}
}
```

为了减少系统开销，在调用 fork()函数后，子进程获得父进程的数据段、内存堆空间及堆栈空间的一个复制。为了实现这种效果，传统设计中，fork 系统调用将父进程的数据段和堆完全复制到子进程的地址空间。但是，在 Linux 和部分 UNIX 的实现中，fork()函数使用了 COW（Copy On Write）技术。子进程并不完全复制父进程的数据，而是采用只读的方式共享父进程的页表，并将父进程的相应页表项也标记为只读。当父子进程中的任何一个进程试图访问这些地址空间时，就会引发系统的页错误异常。异常处理例程将会生成该页的一份复制，并修改进程的页表项，指向新创建的页面，并且将该页标记为已修改。

子进程之间除了共享父进程的数据之外，还继承如下很多父进程的其他属性：

- 身份信息。包括实际用户 ID、实际组 ID、有效用户 ID、有效组 ID 等。
- 文件系统信息。包括当前工作目录、根目录、打开文件描述符等。
- 环境信息。包括用户资源配额限制、信号屏蔽和排列等。

父、子进程之间的不同之处在于：

- fork：系统调用的返回值。
- 进程关系：包括父进程 ID、进程的链接信息、左右兄弟等。
- 进程的统计信息：包括进程的用户态时间、系统态时间等。
- 文件锁：子进程不继承父进程的文件锁，以免引起死锁。
- 待处理信号：子进程不继承父进程的待处理信号（sigpending）

UNIX（Linux）中还提供了 2 个创建任务的系统调用 vfork、clone。和 fork 系统调用一样，vfork 与 clone 都是系统调用的库函数封装。它们的不同之处在于：

- 调用 vfork()函数创建的子进程在调用 exec()函数以前，共享父进程的地址空间，并且父进程将被挂起，直到子进程调用 exec()或是退出执行。
- clone()函数在父进程的地址空间内创建一个新的进程，子进程共享使用父进程的执行环境，包括地址空间、文件描述符表和信号处理例程表等。clone()函数的本质是创建了一个线程，其主要用途在于实现多线程机制。

3.4 常用任务管理 API

1. fork 创建一个新进程

- 功能

创建一个类似父进程的子进程，它与父进程的差别仅在于使用了不同的进程 ID。但事实上新进程的资源利用率被设为 0，并且也不会继承文件锁定以及待处理的信号。Linux 在实现 fork 系统调用时采用了写时复制（Copy On Write）技术，因此，fork 系统调用发生在复制父进程的页表和为子进程创建一个单独的任务结构时，浪费了时间和内存。

- 引用的头文件

<sys/types.h>

<unistd.h>

- 函数原型

pid_t　fork (void)

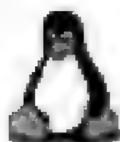
- 返回值

如果创建成功，在父进程中返回的是子进程的 ID，而在子进程中返回值为 0。如果创建失败，则系统会返回-1 给父进程。

- 错误代码
  - EAGAIN fork：不能分配足够的内存来复制父进程的页表和为子进程创建一个单独的任务结构。
  - ENOMEM fork：由于内存紧张，分配必要的内核结构失败。

下面是使用 fork()函数创建子进程的一个例子：

```
main()
{
        int pid;
        pid=fork();
        if(pid>0)
{
        printf("Father !, the child pid is %d\n",pid);
        exit(0);
}
if(pid==0)
{
        printf("Child\n");
        exit(0);
}
if(pid<0)
{
```




```
        printf("Error,Create Failure!\n");
exit(1);
}
}
```

2. clone 按指定条件创建子进程

- 功能

和 fork()函数调用相同的是，clone()函数调用也用于创建一个新进程。不同之处在于，这些调用允许子进程共享它的调用进程的部分执行环境，如内存空间、文件描述符表和信号处理程序表。clone()函数调用的主要作用是执行线程，用于控制一个程序的多个线程同时运行在一个共享内存空间。

clone()函数调用的几个参数：

➢ fn：是在子进程开始执行时调用的一个指向函数的指针。当 fn(arg)函数应用返回时，子进程被终止。Fn()函数返回的整数值是用于子进程的退出码。子进程也可以通过调用 exit 或在收到一个失败信号后明确终止自身。

➢ child_stack：指明由子进程使用的堆栈的位置。由于子进程和它的调用进程可以共享内存，而子进程不可能和其调用进程使用相同的堆栈来运行。因此，其调用进程必须设置用于子进程堆栈的内存空间并将指向此内存空间的指针传递给 clone()函数。

➢ flags：它的低字节包含了子进程终止时传给父进程的信号个数。如果此信号未被指定为 SIGCHLD，那么当父进程使用 wait()来等待子进程终止时必须被指定为_WALL 或_WCLONE 选项。如果没有指定信号，则父进程在子进程终止时得不到信号通知。flags 可由下面的几种常数项通过或操作来得到，以指明调用进程和子进程之间共享哪些资源。

◆ CLONE_PARENT：如果设置了 CLONE_PARENT，新的子进程与其调用进程拥有同一父进程；如果未设置 CLONE_PARENT，那么子进程的父进程就是其调用进程。

注意：子进程终止时，以信号通知的是其父进程，所以如果设置了 CLONE_PARENT 标志，那么被通知的应该是其调用进程的父进程，而不是调用进程本身。

◆ CLONE_FS：如果设置了 CLONE_FS，调用进程和子进程共享同一文件系统信息，包括文件系统的根目录、当前的工作路径和 umask；如果未设置 CLONE_FS，在 clone()调用期间，子进程使用的是其调用进程的文件系统信息的一个拷贝。

◆ CLONE_FILES：如果设置了 CLONE_FILES，调用进程和子进程共享同一文件描述符表。在调用进程和子进程里，文件描述符通常指向同一文件；如果未设置 CLONE_FILES，在 clone 调用期间，子进程继承其调用进程中打开的所有文件描述符的一个拷贝。

- 引用的头文件

<sched.h>

- 函数原型

int clone (int (*fn) (void *), void *child_stack,int flags, void *arg)

- 返回值和错误代码

调用成功，将返回子进程的 pid 给调用线程，如果失败，返回-1 给调用进程。

3. exec 系列函数运行可执行文件

- 功能

exec 系列函数（execl()、execlp()、execle()、execv()、execvp()）用于执行一个新的可执行文件。使用新的进程映像代替当前进程映像，exec 并不创建新进程，所以进程 ID 并不改变。exec 只是用另一个新程序替换了当前进程的正文、数据、内存堆和堆栈段。这些函数其实可以看作 execve()函数的前端，因为都是通过调用 execve()函数实现的。

这些函数的第一参数是将要执行的可执行文件的路径。execl()、execlp()、execle()函数中的参数 arg 和后面的省略号（代表可变参数，可被看作 arg0、arg1、…、argn）描述了传递给该执行程序的参数（都是 NULL 终结字符串）。

execv()、execvp()函数中的参数 argv 表示要传递给新程序的参数数组（都是 NULL 终结字符串）。按照惯例，第一个参数指向将要执行的程序的文件名，而该数组必须以 NULL 指针结尾。

execle()函数另外还提供了 envp 数组表示提供给新程序，同样该数组必须以 NULL 指针结尾。

execlp()、execvp()函数指定以 filename 作为参数时，如果 filename 中包含“/”字符，则就将其视为路径名。否则，就按 PATH 环境变量，在有关目录中搜寻可执行文件。

- 引用的头文件

#include <unistd.h>

- 函数原型

```
int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char *path, const char *arg , ..., char* const envp[]);
int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
```

- 返回值和错误代码

exec 系列函数在正常情况下不会返回。如果任何 exec 系列函数返回，意味着发生了错误。函数的返回值为-1，全局变量 errno 表明了错误的具体类型。

4. exit 终止进程

- 功能

exit()函数用于引起正常的程序终止，status 变量的值将会返回给父进程。所有使用



atexit()函数和 on_exit()函数注册的函数会以注册相反的顺序调用执行。所有打开的流将被刷新，然后关闭。

● 引用的头文件

#include <stdlib.h>

● 函数原型

void exit(int status);

● 返回值和错误代码

exit()函数不会返回。

5. _exit()函数立即终止当前进程

● 功能

_exit()函数用于立即终止调用进程的执行，属于该进程的所有打开的描述符将被关闭。所有该进程的子进程将被1号进程（init）接管，并向该进程的父进程发送 SIGCHLD 信号。

● 引用的头文件

#include <stdlib.h>

● 函数原型

void _exit(int status);

● 返回值和错误代码

_exit()函数不会返回。

6. nice 改变分时进程的优先级

● 功能

nice()函数将调用进程的 nice 值加 inc。注意，根据 UNIX 系统的预定，nice 值越大代表的优先级越低。只有超级用户可以使用负的 inc 值，即提高进程的优先级。

● 引用的头文件

#include <stdlib.h>

● 函数原型

int nice(int inc);

● 返回值和错误代码

调用成功时，返回值为 0；如果调用失败，则返回值为-1。全局变量 errno 表明了错误的具体类型。

7. pause 挂起进程，等待信号

● 功能

pause()库函数调用引起调用进程（或线程）睡眠，直到收到一个信号为止。

● 引用的头文件

#include <stdlib.h>

● 函数原型



int pause(void);

- 返回值和错误代码

pause()函数只在收到一个信号，进入信号处理程序运行完成后，才会返回。在这种情况下，pause 函数返回-1，全局变量 errno 被置为 EINTR。

8. vfork 创建一个子进程

- 功能

在 Linux 中，vfork()函数像 fork()函数一样，用于创建一个子进程。

vfork()函数是 clone()函数的一种特例，它不复制进程的虚拟地址空间。用于子进程创建后，立即调用 exec 函数的情况。该函数对性能敏感的应用非常有用。

vfork()函数与 fork()函数的不同之处在于，调用 vfork()函数的父进程将被挂起，直到子进程调用 exec()函数，或是子进程退出。

- 引用的头文件

```
#include <sys/types.h>
#include <unistd.h>
```

- 函数原型

pid_t vfork(void);

- 返回值和错误代码

其返回值和错误代码的含义可以参考 fork 调用中的介绍。

9. wait()函数等待子进程终止

- 功能

wait()函数用于挂起当前进程的执行，直到它的某个子进程退出，或是收到一个终止当前任务运行的信号，或是一个需要调用信号处理函数的信号。如果在调用该函数的同时，一个子进程已经退出（即所谓的僵死进程），则该函数将立即返回，该子进程占用的全部系统资源将被释放。

- 引用的头文件

```
#include <sys/types.h>
#include <sys/wait.h>
```

- 函数原型

pid_t wait(int *status);

- 参数

pid 参数可以取以下值之一：

  - < -1：等待其组 ID 等于 pid 绝对值的任一子进程。
  - -1：等待任一子进程，这时 waitpid 和 wait 等效。
  - 0：等待其组 ID 等于调用进程的组 ID 的任一子进程。
  - >0：等待其进程 ID 等于 pid 任一子进程。

如果 status 参数不为 NULL，wait()函数会将子进程的状态信息保存到 status 所指向的



地址。用户可以使用下面的宏来取得相关信息。

- WIFEXITED(status)：非 0，子进程正常退出。
- WEXITSTATUS(status)：取子进程返回码的低 8 位。该返回码既可以是作为 exit()函数的参数而设置的，也可以是在主程序中，作为 return 语句的参数而设置。但该宏必须在 WIFEXITED(status)返回非 0 值的时候使用。
- WIFSIGNSLED(status)：返回值为真。子进程的退出，是因为一个未捕获的信号而发生时，将返回该值。
- WTERMSIG(status)：用于返回引起子进程终止的信号编号。该宏必须在 IFSIGNSLED (status)返回真值的时候使用。
- WIFSTOPPED(status)：返回真，如果欲返回的子进程当前已经停止，该情况只会在使用 WNUTRACED 参数调用 wait 函数时发生。
- WSTOPSIG(status)：返回引起子进程停止的信号编号，这个宏必须在 WIFSTOPPED (status)返回真值的时候使用。
- WCOREDUMP(status)：检查子进程是否转储核心。该宏目前只在 Linux、Solaris 平台上实现了。使用该宏之前，应该先使用类似#ifdef WCOREDUMP ... #endif 的代码，检查 WCOREDUMP 是否已定义。

● 返回值和错误代码

正常情况下，返回子进程的 pid，如果使用不等待标志 WNOHANG，则返回 0，当返回-1，表示出错，errno 表明了错误的值。

3.5 关于任务的实例

下面来看一个实际应用程序的源代码，该段代码摘自 ftpd-Linux 下一个著名的 FTP 服务器软件的 uClinux 版本。下面的代码来自 Server_mode.c 文件中的 server_mode()函数。在调用该函数之前，main()函数已经进行了输入参数的处理，然后调用 server_mode()建立服务器链接，等待处理来自网络的 FTP 链接请求。该函数较长，接下来，将分段对它们进行说明。

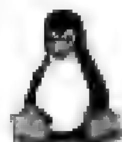
```
int
server_mode (const char *pidfile, struct sockaddr_in *phis_addr)
{
   int ctl_sock, fd;
   struct servent *sv;
   int port;
   static struct   sockaddr_in server_addr;   /* Our address */
   /*声明该函数用到的一些局部变量，包括用于 FTP 控制链接的插口 ctl_sock，使用
accept 函数返回的 socket 链接的文件描述符、服务器环境 sv、端口地址 prot 和服务器地
址 server_addr 等*/
```

```
  /* Become a daemon. */
#ifdef HAVE_DAEMON
  if (daemon(1,1) < 0)  /*调用 daemon 系统调用，将当前进程设置为精灵进程，因为作为一个服务器程序，FTPD 服务进程不需要控制终端，这也是绝大多数网络服务器采取的方式*/
#endif
    {
      syslog (LOG_ERR, "failed to become a daemon");  /*如果设置精灵程序不成功，输出错误记录到系统日志*/
      return -1;
    }
  (void) signal (SIGCHLD, reapchild);  /*安装 SIGCHLD 信号的处理程序，以便在其子进程（具体处理每一个客户链接的进程）结束之后，收集子进程的退出信息。关于信号的操作，将在以后详细介绍*/

  /* Get port for ftp/tcp */
  sv = getservbyname ("ftp", "tcp");  /*为了能在正确的端口上建立 FTP 服务，使用 getservbyname 系统调用，试图取得 FTP/TCP 服务的端口地址，并将结构保存到 sv 结构中*/
  port = (sv == NULL) ? DEFPORT : sv->s_port;  /*检查 sv 的值，如果 sv 为空，说明 getservbyname 没有成功，则使用默认的端口（FTP 服务的控制链接默认端口号为 21），否则使用 sv->sport 作为端口地址*/

  /* Open socket, bind and start listen.  */
  ctl_sock = socket (AF_INET, SOCK_STREAM, 0);  /*调用 socket 系统调用打开了网络插口，并将新创建的插口号返回给 ctl_sock。Socket 调用的第一个参数 AF_INET，代表使用 IPV4 协议，第二个参数 SOCK_STREAM，代表使用串行，基于流的、可靠的有连接协议，该协议支持带外数据传输机制*/
  if (ctl_sock < 0)
    {
      syslog (LOG_ERR, "control socket: %m");
      return -1;
    }          /*检查 socket 调用是否成功执行，如果不成功，则输出错误记录到系统日志*/

  /* Enable local address reuse.  */
  {
    int on = 1;
    if (setsockopt (ctl_sock, SOL_SOCKET, SO_REUSEADDR,
            (char *)&on, sizeof(on)) < 0)  /*调用 setsockopt 系统调用，设置本地地址可以重用*/,
      syslog (LOG_ERR, "control setsockopt: %m"); /*如果出错，输出错误信息*/
```




```
    }
    /*下面设置服务器地址*/
    memset (&server_addr, 0, sizeof server_addr);           /*清零该数据结构*/
    server_addr.sin_family = AF_INET;                       /*设置使用的协议*/
    server_addr.sin_port = htons (port);                    /*设置使用的端口*/

    if (bind (ctl_sock, (struct sockaddr )&server_addr, sizeof server_addr))/*将前面创建的
插口绑定到服务器地址*/
        {
          syslog (LOG_ERR, "control bind: %m");
          return -1;                             /*如果绑定失败，输出错误信息，返回*/
        }
    if (listen (ctl_sock, 32) < 0)   /*开始侦听客户端的链接请求，参数 32，指定了同时最
多能接收的链接数*/
        {
          syslog (LOG_ERR, "control listen: %m");
          return -1;                             /*如果出错，记录错位信息，该函数返回*/
}
```

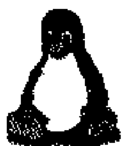
到此为止，网络链接的初始化完全结束。下面的代码中将使用多进程编程模式，对每个客户请求进行处理，这也是要关注的重点。

```
  /*下面，将当前进程的 pid 保存到 pidfile 文件中，并修改该文件的执行权限*/
    /* Stash pid in pidfile*/
    {
     FILE pid_fp = fopen (pidfile, "w"); /*以可写方式打开 pidfile 文件*/
      if (pid_fp == NULL)
        syslog (LOG_ERR, "can't open %s: %m", PATH_FTPDPID);   /*如果打开不成功，
记录错误信息*/
       else
         {
    fprintf (pid_fp, "%d\n", getpid());    /*使用 fprintf()函数，将使用 getpid()系统调用取得
的进程 ID 写入 pidfile 文件*/
    fchmod (fileno(pid_fp), S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH); /*调用 fchmod 系统
调用设置 pidfile 文件的权限为拥有者可读写，组用户和其他用户都只能读*/
    fclose (pid_fp);   /*关闭该文件*/
         }
    }
    /*下面是一个无限循环，在循环体中，一旦侦听到有客户端链接请求后，将调用 fork
或是 vfork 创建一个新的子进程处理该客户请求，而父进程则继续侦听新的客户端链接
请求*/
    /* Loop forever accepting connection requests and forking off
```

```
        children to handle them.   */
    while (1)
      {
        int addrlen = sizeof (*phis_addr);
        fd = accept (ctl_sock, (struct sockaddr )phis_addr, &addrlen); /*在 socket 上侦听客户端链接情况。该函数是阻塞型函数。知道有新的链接请求到来了，该函数调用才返回。下面就是创建子进程处理该请求。因为 uClinux 中，使用 vfork 替代 fork 函数*/
  #ifdef HAVE_WORKING_FORK
        if (fork () == 0) /* child */
  #else
        if (vfork () == 0) /* child */
  #endif      /*根据宏 HAVE_WORKING_FORK 判断使用 fork 还是 vfork 来创建子进程*/
      {   /*下面是子进程的执行代码*/
      (void) dup2 (fd, 0);   /*把接收到的客户连接插口文件描述符复制到子进程的标准输入上*/
      (void) dup2 (fd, 1);   /*把接收到的客户连接插口文件描述符复制到子进程的标准输出上*/
      close (ctl_sock);    /*关闭了控制链接的插口的文件描述符 ctl_sock，这样做既可以方便子进程的操作，又可以防止子进程错误地操作标准输入输出，而引起其他安全问题*/
      break;
      }
        close (fd);  /*父进程代码，该代码关闭了刚接收到的客户连接插口文件描述符，为继续接收新的客户链接做好了准备*/
    /*父进程代码到此为止，父进程将一直在此循环接收新的客户链接，然后创建新的子进程来处理。因为在前面设置了 SIG_CHLD 信号的处理函数。因此，当父进程收到 SIG_CHLD 信号，父进程将调用 reapchild 收集子进程的退出状态。另外，当父进程收到退出信号时，则结束整个程序的运行*/
      }
    /*从现在开始，以后的代码都在子进程空间执行*/
  #ifdef WITH_WRAP
    /* In the child.   */
    if (!check_host ((struct sockaddr *)phis_addr))
      return -1;
  #endif          /*对客户端地址进行检查*/
    return fd;      /*关闭 fd 文件描述符，因为前面已经复制了该描述符*/
  }
```

3.6 小结

任务（进程和线程）是实现多道程序并发执行的必要基础。操作系统必须依照某种调度算法，合理地调度任务运行。操作系统必须给进程分配资源，允许进程共享和交换信息，



保护每个进程的资源以避免其他进程的干扰，允许进程间的同步执行。为达到这些要求，操作系统必须为每个进程维护一个数据结构，用以描述该进程的状态和资源所有权，并允许操作系统行使对该进程的控制权。

3.7 思考题

1. 说明任务、进程、线程的区别和联系。
2. 进程 3 个基本状态及其转变的原因是什么？
3. 请思考任务状态划分的原则。任务状态还可进一步划分吗？
4. 任务调度的常用算法及其目标是什么？
5. Linux 任务调度的优缺点是什么？

第4章　任务的同步与通信

知识点：

- 任务同步的概念与重要性
- 死锁的产生与解决方法
- 优先级反转及其解决方法
- 任务间通信的常用机制
- Linux 基本通信机制与 SystemV IPC

本章导读：

多道程序设计技术在提高资源利用率和系统吞吐量的同时，也引入了新的问题。实现多个相互合作的任务之间的协调一致地共同运行，需要有新的机制和方法来提供保证。多个任务间的关系可以分为同步和通信两种。本章将介绍任务的同步与互斥的基本概念，以及任务间通信的常见问题和相应的解决方法。



4.1 任务间同步与互斥

在嵌入式多任务操作系统中，多任务的引入改善了系统的资源利用率，并且提高了系统的吞吐量，但是也带来了另外的问题，那就是多个任务间如何协调、合作共同完成一个大的系统功能。特别是当它们在竞争使用临界资源，或是需要相互通知某些事件发生时。例如，多个任务去竞争使用同一台打印机时，有可能使多个任务的打印结果交织在一起，造成混乱。而多个相互合作的任务，比如在工控系统中，可以将数据采集、数据处理和数据输出划分为不同的任务，它们之间需要一种通信机制，使得采集任务可以通知数据处理任务新采集的数据已经到达，而数据处理任务可以通知输出任务，需要输出的结果已经计算完成。

由上所述，系统中的多个任务间可能存在 3 种关系。

1. 资源共享

多个任务间没有直接联系，它们并不知道其他任务的存在。但是，这些任务在运行时，都会使用某些公共的资源，而这些资源往往数量有限，甚至只能独占使用。因此，这些资源不能允许用户随意使用，而应该使用一种机制，保证对资源的使用是满足系统的限制条件的。常见的机制有：锁、信号量等。资源共享又分为两种情况：

- 数据共享：应用程序使用多个任务，同时并行处理一组数据时，为了避免重复，应该提供某种互斥机制，限制同时只有一个任务访问该组数据。这种情况在并行计算中大量存在。例如，多任务下载，Web 请求的处理等。
- 共享外部设备：多个应用程序同时访问独占性的外部设备。例如，在只有一台打印机的系统中，如果打印机已经分配给任务 1 使用，在任务 1 使用打印机的过程中，其他任务的打印申请都得不到满足。只有当任务 1 完成打印，并释放打印机后，系统才能将打印机分配给其他任务使用。

2. 相互合作

为了完成某些大型应用程序，程序员往往将程序分为多个任务，每个任务完成一种功能，多个任务间必须相互合作，才能完成系统的全部功能。例如，在上面提到的数据采集、处理、输出系统中，就是一种相互合作的关系。此时，任务间的同步机制，必须保证相互合作的多个任务在执行次序上的协调，即时间关系上，多个任务应该按部就班地向前执行。例如，首先数据处理任务必须等待采集任务收集到足够的数据，然后才能执行，而输出任务必须等待数据处理任务计算完成，才能对外输出结果。

3. 相互无关

在相互无关情况下，任务之间没有任何联系，它们既没有共享的资源，也没有时间、空间和功能上的协作，这种任务可以分别独立运行。

4.2 任务间的同步

4.2.1 重要概念

1. 临界资源与临界区

把在一段时间内只允许一个任务访问的资源叫做临界资源（critical resource），即当该资源已被某个任务占用时，新的要使用该资源的申请，必须等到前一个任务完成并释放该资源后，才能执行。在前面提到的共享数据和共享外部设备很多都是临界资源，比如打印机、磁带机等。

把程序中使用临界资源的那段代码称为临界区（critical section）。为了实现对临界资源的互斥访问，在进入临界区以前，必须检查该资源当前是否正被访问。如果此刻临界资源未被访问，该任务便可以进入临界区，并将其设置为被访问状态。然后，即可以对临界资源进行操作。操作完成后，该任务退出临界区，将其访问标志清除，释放对临界资源的占有，其他请求使用该资源的任务便可以再次进入临界区。

2. 原子操作

任务间步就是找到一种方法，确保一个任务在使用一个共享资源时，其他任务不能访问相同的资源，这实质上就是互斥。为了实现互斥，历史上众多的数学家和计算机科学家进行了尝试，提出的解决方案可以分为两类：一类是软件方法；一类是硬件方法。软件解决方法包括：

- 关闭中断或关闭调度。
- 使用锁变量。
- 轮换法。
- Peterson 方法。

上述的软件解决方法都有忙等待的缺点。软件方法解决同步问题的困难在于实现对数据的原子操作。因此，为了更有效地实现互斥访问，现代的微处理器提供专用的原子操作指令。

最常见的原子操作指令是 TSL（测试并上锁）指令。TSL 指令的功能如下：首先，将内存中一个字读入一个寄存器中；然后，将原内存地址写入一个非零值。整个操作具有原子性，是不可分割的，即读和修改操作必须同时进行，要么都执行，要么都不执行。TSL 指令功能可以使用下面的函数来描述：

```
bool TSL(const bool lock)
{
        bool temp;
        temp = lock;
```




```
        lock = true;
        return temp;
}
```

使用 TSL 指令来实现互斥的方法如下：

初始化时可根据资源是否可使用，将变量 lock 置为 true 或 false。然后在进入临界区前，使用 TSL 指令检查 lock 是否为 true，lock 为 true 表示当前临界资源正在使用中，则空循环等待，否则进入临界区进行操作。当任务退出临界区时，将 lock 变量重新赋值为 false，以便允许其他任务进入临界区。该算法的伪代码描述如下：

```
bool lock=false;
...
while(TSL(lock))
{
        ;
}
...
临界区代码
...
lock = false;
```

另外一条常用于原子操作的指令是 SWAP/XCHG（对换）指令，使用该指令可以交换内存空间中两个地址的内容。该指令的功能用伪代码可以描述如下：

```
void SWAP(int a, int b)
{
        int temp;
        temp = a;
        a = b;
        b = temp;
}
```

使用 SWAP/XCHG 指令来实现互斥的方法如下：

初始化时将变量 lock 置为 false。另外，使用一个标志变量 flag，在进入临界区前，将 flag 赋值为 true，再使用 SWAP 指令交换 lock 与 flag 的值，然后检查 flag 是否为 true。flag 为 true，表示当前临界资源正在使用中，则空循环等待；否则，程序进入临界区进行操作。当任务退出临界区时，将 lock 变量重新赋值为 false，以便允许其他任务进入临界区。该算法的伪代码描述如下：

```
lock = false;


...
flag = true;
do
{
        SWAP(lock,flag);
}
while(flag == true)
```

3. 死锁

在多道程序系统中，多个任务因为竞争资源而循环等待的情况，叫做死锁。多个竞争共享资源的任务执行顺序不当，或是代码编写有问题时，就有可能引起死锁，下面是一个死锁的例子。

假设系统中有两个任务（t1、t2），它们需要共享使用两种资源（r1、r2），这两种资源分别使用两个信号量（s1、s2）来实现互斥访问。下面是这两个任务的伪代码：

任务 1　　　　　　　　　　　　任务 2

```
Begin                      Begin
P(s1);                     P(s2);
Operation on r1;           Operation on r2;
P(s2);                     P(s1);
Operation on r2;           Operation on r1;
V(s2);                     V(s1);
V(s1);                     V(s2);
End;                       End;
```

在特定的情况下，这两个任务间可能产生死锁，例如下面这种情况。

首先 t1 任务调用 P(s1)，申请占有信号量 s1，该操作将会成功；然后 t1 对资源 r1 进行操作。假设此时 t1 任务的时间片正好用完，系统将调度 t2 任务执行，t2 任务在执行过程中调用 P(s2)，申请占有信号量 s2，该操作成功。

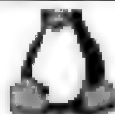
接下来 t2 任务对资源 r2 进行操作；然后 t2 任务调用 P(s1)，申请占有信号量 s1，由于信号量 s1 此时已被 t1 任务占有，因此，t2 任务不能继续执行，P 操作将 t2 任务挂起。

于是，调度器调度 t1 任务执行，t1 任务接着调用 P(s2)，申请占有信号量 s2，但信号量 s2 已被 t2 任务占用，因此，t1 任务也不能继续执行，P 操作将 t1 任务挂起。

现在，t1 任务和 t2 任务相互等待，都不能继续运行，若无外力作用，这种死锁状态将永远保持下去。

4. 死锁产生的必要条件

死锁的产生必须具有以下 4 个必要条件：



- 资源的互斥访问条件。资源的访问是独占性的，即同时最多只能有一个任务访问该资源。
- 资源逐次申请条件。任务可以申请多个资源，当申请的某个资源被其他任务占用时，任务被阻塞，但是并不释放已经占用的资源。
- 资源不可剥夺条件。任务已经申请使用的资源，在使用未完成前，不能被剥夺，只能由占用者主动释放。
- 环路等待条件。发生死锁时，必然存在一个由三元组（任务、占有资源和申请资源）组成的环。如以下的三元组序列：

```
(t1,r1,r2),
(t2,r2,r3),
...
(tn,rn,r1),
```

该序列代表任务 1 占有资源 1，申请资源 2；任务 2 占有资源 2，申请资源 3；如此，直到任务 n 占有资源 n，申请资源 1。

5. 死锁的解决

死锁的解决办法可以分为 3 类：死锁预防、死锁避免、死锁检测和解除。死锁预防主要是通过消除产生死锁必要条件之一以预防死锁的产生。死锁避免是指在系统运行中，在资源分配之前，检查该次分配是否可能引起死锁，如有可能就进行相应的处理以达到目的。死锁检测和解除的基本思想是在系统运行过程中，检查当前是否形成死锁，如果是，则放弃等待环路某个等待的任务，让其他任务继续运行下去，而解除死锁。

6. 优先级反转

按照优先级驱动的设计意图，低优先级的任务不应该阻塞高优先级任务的运行。通常，将由低优先级的任务阻塞高优先级任务执行的异常情况称为优先级反转。优先级反转是一种不确定的延迟现象，经常出现于存在共享资源的多任务、可抢占的执行体中。当高优先级任务企图访问已被某低优先级任务占有的共享资源时，就会引起优先级反转。高优先级任务必须等待直到低优先级任务释放它所占有的资源。如果该低优先级任务又被一个或多个中等优先级任务阻塞，问题就更加严重。因为低优先级任务得不到执行就不能访问资源、释放资源。于是，低优先级任务就以一个不确定的时间阻塞高优先级的任务，系统的实时性就没有保障。图 4-1 所示是一个优先级反转的示例。

在图 4-1 中系统存在任务 A、B、C（优先级从高到低排列）和资源 D。某时刻，任务 A 和任务 B 都被阻塞，任务 C 运行，占用资源 D。一段时间后，任务 A 和任务 B 相继就绪，任务 A 抢占任务 C 运行，在申请资源 D 时失败而被挂起。由于任务 B 的优先级高于任务 C，此时任务 B 开始运行。此时，任务 A 优先级最高，但是因为任务 C 占用资源 D 而被阻塞。而任务 B 优先级高于任务 C，因此可以执行。因此，任务 A 的执行就被不确定地延迟了（依赖于任务 B 执行时间）。在极端情况下，任务 A 永远无法运行，将处于饿死状态。

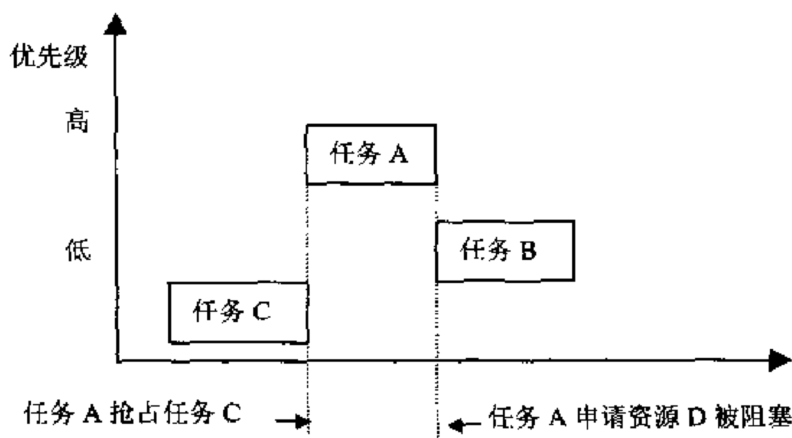


图 4-1 一个优先级反转的示例

产生优先级反转的根本原因是，资源访问的独占性和无特权性与任务调度的可抢占性和有特权特性相冲突。目前，解决优先级反转问题有两种经典方法——优先级继承算法和优先级天花板算法。

（1）优先级继承。

优先级继承是指将低优先级任务的优先级提升到等待它所占有的资源的最高优先级任务的优先级。每当高优先级任务由于等待资源而被阻塞时，此时资源的拥有者的优先级将会自动被提升。

互斥信号量可以采用优先级继承算法。当占有互斥信号量的任务的优先级低于请求获得互斥信号量的任务的优先级时，占有互斥信号量的任务的优先级将被提升到请求互斥信号量的任务的优先级。当任务释放完它所占有的全部互斥信号量时，它的优先级才恢复到它原有的优先级。图 4-2 所示是优先级继承的一个示例。

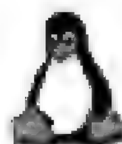
如图 4-2 所示，使用优先级继承后，任务的调度顺序如下：当任务 A 因申请资源 D 而被挂起时，将提升任务 C 的优先级，使它的优先级等于任务 A 的优先级。提升后的任务 C 不再被中间优先级任务抢占，从而确保它能尽快释放资源 D，让任务 A 得以运行。任务 C 释放出资源 D 后，其优先级将恢复为原来的优先级。

（2）优先级天花板。

优先级天花板算法是指将申请某资源的任务的优先级提升到可能访问该资源的所有任务中最高优先级任务的优先级，这个优先级称为该资源的优先级天花板。互斥信号量的优先级天花板在它被创建时指定。图 4-3 所示是优先级天花板的一个示例。

如图 4-3 所示，使用优先级天花板算法后，任务的调度顺序如下（假定任务 A 为申请资源 D 的所有任务中优先级最高的任务）：当任务 C 申请资源 D 时，优先级被提升至任务 A 的优先级。提升后的任务 C 不再被中间优先级的任务抢占，从而确保它能尽快释放资源 D，让任务 A 得以运行。任务 C 释放出资源 D 后，将恢复为原来的优先级。

这两种算法对任务优先级的改变在一定程度上都会影响应用中预先设定的任务流程。对于优先级继承算法，只有在占有资源的低优先级任务阻塞高优先级任务成为事实时才会



采取相应的措施；而对于优先级天花板算法则不论是否发生阻塞都要采取相应的措施。所以相对而言，优先级继承对应用中任务流程的影响要小于优先级天花板。对于这两种算法都支持的内核，应用程序应该选择哪种算法来解决优先级反转问题比较合适，应视实际情况而定。

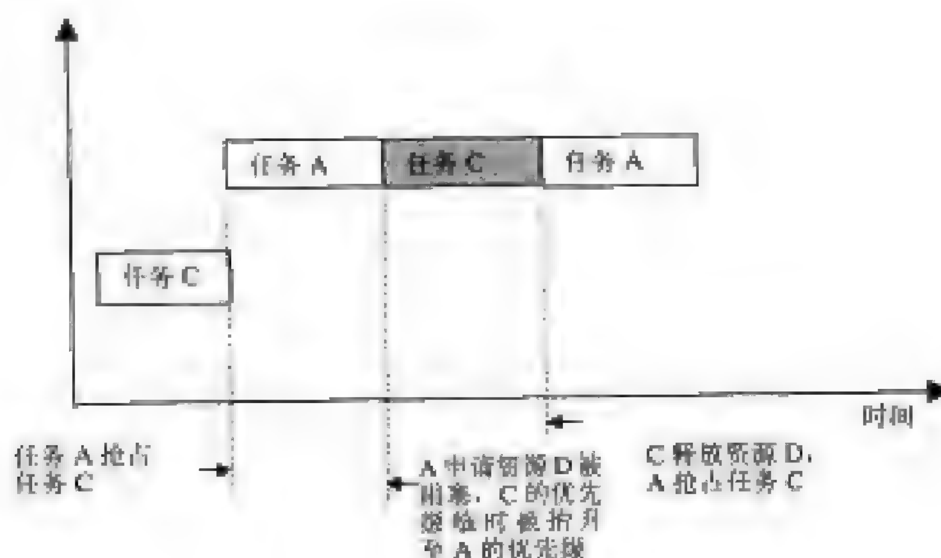


图 4-2　一个优先级继承的示例

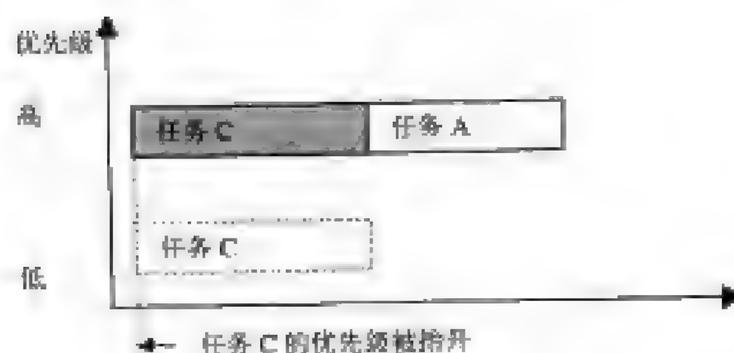


图 4-3　一个优先级天花板的示例

本小节介绍了任务同步中的几个问题，下面将介绍基于信号量机制的几个解决方法。

4.2.2　信号量的概念

1. 信号量机制

1965 年 Dijkstra 提出的信号量，是目前广泛得到应用的任务同步和互斥方法。它使用

一个整型变量来表示资源的数目，使用两个原子操作 P 和 V 来对信号量进行加减操作。当信号量大于零时，表示当前资源尚有空闲，申请使用该信号量将成功；当信号量等于零时，表示当前资源已经被占用，申请使用该信号量的任务将被阻塞。

原始的 P、V 操作可以用伪代码描述如下：

```
void P(int s)
{
    while(s<=0)
{;}
              s--;
}

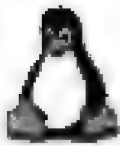
void V(int s)
{
    s++;
}
```

原始的整型信号量提供了任务间互斥访问资源的能力，但是仍然有忙等待的缺点。如果要解决忙等待的问题，必须要有操作系统的支持。其思想如下，在 P 操作中，若发现所申请的资源已被占用时，则通过系统调用 Block，将执行 P 操作的任务挂起；在 V 操作中，若发现资源已经空闲时，则通过系统调用 UnBlock，将正在等待该资源的任务唤醒。为了能够区分等待不同信号量的任务，应该为每个信号量设置一个 block 队列，以便将因申请该信号量而阻塞的任务挂入该队列。操作系统教材把这种信号量机制叫做记录型信号量或者计数信号量。计数信号量操作可以用伪代码描述如下：

```
struct semaphore
{
        int count;
  task_wait_queue   wait;
}

void P(semaphore s)
{
s.count--;
if(s.count<0) then
        Block(Self,s.wait);
}

void V(semaphore s)
{
```




```
s.count++;
if(s.count<=0) then
        UnBlock(s.wait);
}
```

其中 Block 和 UnBlock 是各种系统调用的抽象，Block(Self,s.wait)表示将当前执行 P 操作的任务自身阻塞起来，加入到 s 的等待队列中。UnBlock(s.wait)表示从 s 的等待队列中唤醒一个任务。

使用这种方法，可以实现任务的"让权等待"。目前很多嵌入式实时操作系统都提供了这种"让权等待"的计数信号量机制。但是，计数信号量机制也有自身的缺陷，主要体现在：

- 单个计数信号量只能解决一个临界资源的互斥访问，使用多个计数信号量不当有可能引起死锁。
- 计数信号量的使用可能引起优先级反转。

2. 信号量集合

死锁预防的一种思想就是让任务在开始运行时，就申请占用所有将要使用的临界资源。基于这种思想，System V（一种 UNIX 发行版）提出了信号量集合的机制。在信号量集合机制中，信号量申请的基本单位不再是单个的信号量，而是一个信号量的集合。需要访问临界资源的任务，首先申请占用该信号量集合，此时系统进行检查，只要申请的资源中任何一个还不能分配，则该信号量集合都不能分配，即使其他资源都已就绪。直到所有欲申请的资源都空闲了，申请该信号量的操作才能成功。当任务的临界操作完成后，将一次性释放所有的资源。下面是信号量集合机制的伪代码描述：

```
    P(Semaphore    S1,S2,...,Sn)
    {
        while(1)
        {
if (S1.count >=1) && (S2.count>=1)&& ... &&(Sn.count>=1)
        {
        for(i=0;i<n;i++)
                        Si.count--;
                break;
        }else
            {
    Block task into wait queue associated with first Si which Si.count<1
}
        }
}

    V(Semaphore    S1,S2,...,Sn)
```

```
{
for(i=0;i<n;i++)
            Si.count++;
    Unblock all task waiting on wait queue associated with all Si
}
```

传统 UNIX 操作系统（System V）和 Linux 都提供了类似于信号量集合的机制。

4.3 任务间的通信

任务间通信是指进程之间的信息交换。任务间的同步与互斥可以看作是传递的信息量较少的一种通信方式。目前常见的任务通信方式可以分为 3 大类：共享存储器、报文传递系统和管道通信。

1. 共享存储器

支持共享存储器的系统，允许多个（两个以上）进程共享给定的存储区域。在任务进行通信前，任务 A 可以向系统申请创建一个共享存储区，若创建成功，系统将返回共享存储区的标识符给任务 A。所有任务都可以使用该标识符把该共享存储区链接到自己的地址空间内，此后，任务就可以像操作普通存储区一样地访问共享存储区。共享存储区是一种高级任务通信机制。它提供了任务间快速交换信息的方法。

2. 报文传递系统

在报文传递系统中，报文是信息交换的基本单位。报文是一个应用相当广泛的概念，在计算机网络系统中，报文一般由前缀段、数据段、后缀段组成。而且报文可以复合，即一个报文的数据段内容可以是一个更上层报文的整体。

按照报文传递的方式，报文传递系统可以分为：

- 直接通信方式：发送任务直接将报文发送给接收任务，并将它挂在接收任务的报文缓冲队列上，接收任务从报文队列中取得报文。
- 间接通信方式：发送任务将报文发送到某种中间实体中，接收进程从中取得报文。这种中间实体一般称为信箱，故这种通信也称为邮箱通信方式。这种模式也被广泛地用于计算机网络通信中。

按照报文的长度，报文传递系统可以分为：

- 定长报文：每次发送的报文长度都是固定的，这种系统实现起来简单，效率较高。
- 变长报文：发送的报文长度可以是用户指定的，这种系统使用灵活、方便。

在直接通信的报文系统中，通常系统提供如下两条通信原语：

- Send(Receiver,message)：表示向 Receiver 发送一个 message 报文。
- Receive(Sender,message)：表示从 Sender 处接收一个报文，存放到 message 中。

而在高级报文系统中，为了让一个任务可以接收多个任务发送的消息，在调用 Receive



原语时 Sender 参数可以为一个空值，表示不特别指定发送进程。而一个任务也可以向多个任务发送消息，这种方式又叫做广播。在广播方式下，调用 Send 原语时 Receiver 可以为一个空值，表示向系统中所有任务发送消息。

在间接通信的报文系统中，通常系统提供如下两条通信原语：

- Send(mailbox,message)：表示向指定邮箱发送一个 message 报文。
- Receive(mailbox,message)：表示从指定邮箱接收一个报文，存放到 message 中。

3. 管道通信

管道通信是 UNIX 系统中最古老、最为重要的一种通信方式。顾名思义，管道提供了一个读进程和一个写进程之间的链接，写（发送）进程以字符流的形式将大量数据送入管道，而读（接收）进程可以从管道中接收数据。

为了通信的正常进行，管道通信机制必须保证：

- 同步：当写进程向管道中发送的数据量超过管道的缓冲区大小时，写进程必须等待管道缓冲区清空。
- 互斥：当多个进程同时向管道读/写数据时，必须保证操作的互斥性。

UNIX 系统中，原始的管道有以下两个限制：

- 管道是半双工的，数据只能在一个方向上流动。
- 管道利用文件描述符管理，只能在具有公共祖先的进程之间使用。

通常，管道由父进程创建，然后，父子进程之间、子进程与子进程之间，就可以通过该管道进行通信。

然而，在 UNIX 系统的一些较新的实现中，FIFO 命名管道、流管道则突破了上面的两个限制，比如 FIFO 管道可用于任意进程间的通信，流管道则可以实现双工通信等。

4.4 嵌入式 Linux 中的任务间同步与通信

相对来说，嵌入式 Linux 中的通信方式就多样化一些。除了传统的管道和信号之外，Linux 也支持 System V 的 IPC 机制，包括共享内存、信号量和消息队列。

4.4.1 Linux 中的信号

信号机制与中断机制非常相似，所以信号又被称为软件中断。在 Linux 中，很多重要的应用程序都要处理信号，从而信号提供了一种处理异步事件的方法。

在 UNIX 的早期版本就有了信号机制，它用来向一个或多个进程发送异步事件信号。UNIX 为常见的信号定义了信号名，这些名字都是以 SIG（signal）开头。比如，SIGABRT 是终止信号。当进程调用 abort()函数时，将产生这种信号。SIGCHID 是子进程状态改变信号，当任务调用 exit()函数，或任务执行完成时，都将会产生这种信号。

Linux 既支持 POSIX.1 的全部信号，也支持 SUSV2（一种 UNIX 标准）的信号和其他一些信号。Linux 内核的信号是这些标准的一个超集。

1. 信号的本质

信号是在软件层次上对中断机制的一种模拟，在效果上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。信号与中断的不同在于信号的实时性不如中断强，中断总是可以打断任务的执行（开中断的情况下），而信号的处理必须是在特定的时间点上。

信号是进程间通信机制中惟一的异步通信机制，可以看作是异步通知，以通知接收信号的进程有哪些事情发生了。信号机制经过 POSIX 实时扩展后，功能更加强大，除了基本通知功能外，还可以传递附加信息。

2. 信号产生的时机

很多条件都可以产生一个信号。信号主要来源可以分为硬件来源和软件来源。

（1）硬件来源。

硬件来源包括硬件异常、除数为零、存储器访问越界、电源失效等，通常这些信号由内核产生，并发送到相关的进程。例如，键盘终端产生的控制信号（如在终端敲击 Delete 键）通常产生终端信号（SIGINT）。

（2）软件来源。

当某些软件事件发生时，也可能产生信号。比如 SIGALRM（进程所设置的闹钟时间已经超时）、SIGPIPE（在管道的读进程已终止后一个进程写此管道）等。

进程使用 kill()和 raise()等函数，可以发送指定的信号给指定的进程和进程组。当然，这对发送进程有一定的权限要求。比如，发送进程必须和接收进程有相同的所有者，或者发送进程的所有者是超级用户。

用户也可以使用 kill 命令发送指定的信号给指定的进程。同样，使用 kill 命令对用户也有一定的权限要求（与 kill()函数相同）。

3. 信号的响应方式

进程可以通过 3 种方式来响应一个信号：

- 忽略信号。即对信号不作任何处理，但其中有两个信号不能忽略：SIGKILL 和 SIGSTOP。
- 捕捉信号。定义信号处理函数，当信号发生时，将执行相应的处理函数。
- 执行默认操作。Linux 对每种信号都规定了默认操作，详细情况可以参考 W.Richard Stevens 的《UNIX 环境高级编程》以及其他相关资料。

注意：进程对实时信号的默认反应是进程终止。

4. 信号列表

前面已经提到 Linux 支持多种标准的信号，在表 4-1 中已经详细列出这些信号。其中，



信号的具体含义将根据操作系统的不同而有所不同。表 4-1 中所列出的信号含义，均以 Linux 为标准。表 4-2 列出的是在 SUSv2 标准中定义了，但是 POSIX.1 标准中没有定义的信号。表 4-3 是在 Linux 自身定义的一些信号（来自 SVR4 和 4.3+BSD）。

表 4-1 在 POSIX.1 标准中定义的信号

信号名	值	动作	说明
SIGHUP	1	A	控制终端挂起或是死亡
SIGINT	2	A	键盘中断
SIGQUIT	3	C	终端退出
SIGILL	4	C	非法指令
SIGABRT	6	C	异常终止 abort()
SIGFPE	8	C	浮点异常
SIGKILL	9	AEF	终止信号
SIGSEGV	11	C	非法内存引用
SIGPIPE	13	A	断开的管道，向没有接收者的管道里写入
SIGALRM	14	A	定时器时间到 alarm(2)
SIGTERM	15	A	终止信号
SIGUSR1	30,10,16	A	用户定义信号
SIGUSR2	31,12,17	A	用户定义信号
SIGCHLD	20,17,18	B	子进程停止或终止信号
SIGCONT	19,18,25		继续执行
SIGSTOP	17,19,23	DEF	停止进程
SIGTSTP	18,20,24	D	中断输入停止符
SIGTTIN	21,21,26	D	后台从控制 tty 读
SIGTTOU	22,22,27	D	后台向控制 tty 写

注：A：默认动作是终止进程；B：默认动作是忽略该信号；C：默认动作是终止进程，内存转储；D：默认动作是停止进程；E：信号不能被捕获；F：信号不能被忽略。

表 4-2 SUSv2 标准增加的信号

信号名	值	默认动作	说明
SIGBUS	10,7,10	C	总线错，即非法内存访问
SIGPOLL	A		SysV 可轮询事件，SIGIO 的同义词
SIGPROF	27,27,29	A	梗概定时器（profiling timer）时间超时
SIGSYS	12,-,12	C	无效的系统调用参数
SIGTRAP	5	C	陷阱断点
SIGURG	16,23,21	B	插口紧急情况（4.2BSD）
SIGVTALRM	26,26,28	A	虚拟时钟（4.2BSD）
SIGXCPU	24,24,30	C	超过 CPU 时间限制（4.2BSD）
SIGXFSZ	25,25,31	C	超过文件大小限制（4.2BSD）

表 4-3 Linux 增加的信号

信号名	值	默认动作	说明
SIGIOT	6	C	IOT 陷阱，等同于 SIGABRT
SIGEMT	7,-,7		硬件故障
SIGSTKFLT	-,16,-	A	协处理器堆栈错
SIGIO	23,29,22	A	I/O 有效信号（4.2 BSD）
SIGCLD	-,-,18		等同于 SIGCHLD
SIGPWR	29,30,19	A	电源失效（System V）
SIGINFO	29,-,-		等同于 SIGPWR
SIGLOST	-,-,-	A	文件锁丢失
SIGWINCH	28,28,20	B	窗口（网络）大小改变（4.3 BSD，Sun）
SIGUNUSED	-,31,-	A	未使用

各种信号的具体含义可以参考 W.Richard Stevens 的《UNIX 环境高级编程》，第 200~203 页。

5. 信号发送的限制

并不是系统中每个进程都可以向其他所有进程发送信号，只有核心和超级用户才具有此权限。普通进程只能向具有相同 uid 和 gid 的进程或者在同一进程组中的进程发送信号。

6. 信号的种类

可以从两个不同的分类角度对信号进行分类：一方面是可靠性，可靠性包括可靠信号与不可靠信号；另一方面是与时间的关系，包括实时信号与非实时信号。

（1）可靠信号与不可靠信号。

Linux 信号机制基本上是从 UNIX 系统中继承过来的。早期 UNIX 系统中的信号机制比较简单和原始，在后来的实践中暴露出一些问题，因此，把那些建立在早期机制上的信号叫做"不可靠信号"，信号值小于 SIGRTMIN（Linux2.4 中，SIGRTMIN=32，SIGRTMAX=63）的信号都是不可靠信号。不可靠信号的主要问题是信号可能丢失。

早期 UNIX 下进程每次处理信号后，就将对信号的响应设置为默认动作（为了避免信号频繁到达引起进程堆栈溢出）。用户如果不希望这样的操作，那么就要在信号处理函数中再一次调用 signal()函数，重新安装该信号，但是这种工作方式是有问题的。虽然，在大多数情况下，这种机制可以工作得很好，但是在某些特殊情况下，该机制可能产生非期望的结果。下面用一个例子来说明这个问题。

```
int sig_int();                                  //信号处理例程
main()
{
```




```
        signal(SIGINT,sig_int);         //安装信号处理例程
        ...
}
int sig_int()
{
        signal(SIGINT,sig_int);         //重新安装信号处理例程
        ...
}
```

该例子试图截获 SIGINT 信号，以便防止用户误操作引起的错误退出，或是在程序退出之前，进行一些扫尾工作。在主程序中，使用 signal()函数捕捉 SIGINT 信号，而在信号处理例程的开始，就立即重新安装该信号处理例程。但是，由于信号的到来是完全随机的，因此，如果在第一个信号到来时，系统将该信号的处理函数设为默认值之后，第二个信号立即到来，此时，系统将使用默认方式对 SIGINT 信号进行处理。因此说，早期 UNIX 对信号的处理是不可靠的。下面再通过一个例子来解释信号的丢失问题。

以下是使用早期信号机制的经典代码：

```
int sig_int();                          //信号处理例程
int sig_int_flag;                       //信号标志
main()
{
        signal(SIGUSR1,sig_int);        //安装信号处理例程
        while(sig_int_flag ==0)
              pause();
        ...
}
int sig_int()
{
        signal(SIGUSR1,sig_int);        //重新安装信号处理例程
        sig_int_flag =1;                //设置主程序的标志，以允许主程序接着运行
}
```

该例子试图使用信号机制提供任务间的同步功能。在主程序中，首先使用 signal()函数捕捉 SIGUSR1 信号；然后检查标志 sig_int_flag 是否被赋值为 1，如果不是，表示等待的信号还没有到达，这时则调用 pause()函数等待一个信号的到达，在信号处理例程中，重新捕捉该信号，然后设置一个标志，表示该用户自定义信号已经发生。但是，在主程序中，检查 sig_int_flag 之后和调用 pause()之前有一段时间空隙，如果信号在这段时间内到达，则该信号将丢失，进程将挂起，直到下一个 SIGUSR1 信号到来。

从上面两个例子可以看出，早期 UNIX 的信号处理方式是信号机制不可靠的根源。Linux 支持不可靠信号，但是对不可靠信号机制作了如下改进：

在调用完信号处理函数后，系统不会将对该信号的处理恢复为默认动作。因此，用户不必每次处理信号后，都重新安装该信号的处理函数。

因此，Linux 下的不可靠信号问题主要指的是信号可能丢失。在当前的 Linux 实现中，0~31 的前 32 个信号都是不可靠信号，32~63 的信号是可靠信号。可靠信号机制支持排队，即多个信号同时到达时，内核会依次记住所有的信号，这样就彻底解决了信号丢失问题。

（2）实时信号与非实时信号。

信号值位于 SIGRTMIN 和 SIGRTMAX 之间的信号都是可靠信号，可靠信号克服了信号可能丢失的问题。为了保持与以前系统的兼容性，Linux 在支持新的信号安装函数 sigation() 以及信号发送函数 sigqueue()的同时，仍然支持早期的 signal()信号安装函数和信号发送函数 kill()。

事实上，signal()和 kill()函数也可以用于安装和发送实时信号，目前 Linux 中的 signal() 函数是通过 sigaction()函数实现的。因此，即使通过 signal()函数安装的信号，在信号处理函数中也不必再调用信号安装函数。同时，由 signal()函数安装的实时信号支持排队，同样不会丢失。Signal()和 sigaction()最大区别在于，经过 sigaction()函数安装的信号都能传递信息给信号处理函数（对所有信号这一点都成立），而经过 signal()函数安装的信号却不能向信号处理函数传递信息。对于信号发送函数来说也是一样的。

7. 信号机制系统调用

Linux 中用于发送信号的主要函数有：kill()、raise()、 sigqueue()、alarm()、setitimer() 以及 abort()等。

（1）kill ()函数终止一个进程。

- 功能

kill()用于发送指定的信号给特定的进程或进程组。如果没有指定信号，将发送 TERM 信号，该信号会杀死那些没有捕捉该信号的进程（如果发送进程拥有相应的权限）。对于其他一些进程，可能需要使用 9 号信号（Kill -9），因为 TERM 信号不能被捕获。

- 引用的头文件

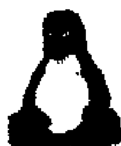
```
#include <sys/types.h>
#include <signal.h>
```

- 函数原型

```
int kill(pid_t pid,int signo);
```

- 参数说明
  - pid：信号的接收进程。
  - pid>0：进程 ID 为 pid 的进程。
  - pid=0：同一个进程组的进程。
  - pid<-1：进程组 ID 为 -pid 的所有进程。
  - pid=1：除发送进程自身外，所有进程 ID 大于 1 的进程。
  - signo：发送的信号编号。

signo 为 0 时（即空信号），实际并不发送任何信号，但照常进行错误检查。因此，可



用于检查目标进程是否存在，以及当前进程是否具有向目标发送信号的权限等。值得注意的是 root 权限的进程可以向任何进程发送信号，非 root 权限的进程只能向属于同一个 session 或者同一个用户的进程发送信号。

● 返回值和错误代码

调用成功返回 0；否则，返回-1，并且将赋值 errno。

（2）raise()函数向当前进程发送一个信号。

● 功能

向当前进程发送一个信号，等价于调用 kill(getpid(),sig)。

● 引用的头文件

#include <signal.h>

● 函数原型

int raise(int sig)

● 参数说明

sig：发送的信号编号。

● 返回值和错误代码

调用成功返回 0；否则，返回-1。

（3）sigqueue()函数向指定进程发送一个信号。

● 功能

发送 sig 指定的信号到 pid 指定的进程，发送信号的权限要求与 kill()相同，并且也可以使用信号 0 来检查该任务是否存在。并且该函数可以使用参数 val 向信号处理例程传递附加信息。但 sigqueue()只能向一个进程发送信号，而不能发送信号给一个进程组。

● 引用的头文件

#include <sys/types.h>

#include <signal.h>

● 函数原型

int sigqueue(pid_t pid, int sig, const union sigval val)

● 参数说明

- pid：信号的接收进程。
- sig：发送的信号编号。
- val：和信号一起发送的附加信息，val 参数的定义如下：

```
union sigval {
            int     sival_int;
            void *sival_ptr;
    }
```

● 返回值和错误代码

调用成功返回 0；否则，返回-1。

（4）alarm()函数为当前进程设置时钟信号。

● 功能



alarm()函数是专门为 SIGALRM 信号而设的，在指定的时间 seconds 秒后，将向进程本身发送 SIGALRM 信号，它又称为闹钟时间。进程调用 alarm()函数后，任何以前的 alarm()函数调用都将无效。如果参数 seconds 为零，那么进程内将不再包含任何闹钟时间。

- 引用的头文件

#include <unistd.h>

- 函数原型

unsigned int alarm(unsigned int seconds)

- 参数说明

seconds：定时器信号的超时时间。

- 返回值和错误代码

如果调用 alarm()函数前，进程中已经设置了闹钟时间，则返回上一个闹钟时间的剩余时间，否则返回 0。

（5）setitimer()函数为当前进程设置时钟。

- 功能

设置 which 指定的时钟的值为 value，如果 ovalue 指针不为 NULL，则将原来的值保存在 ovalue 所指向的结构中。

- 引用的头文件 1

#include <sys/time.h>

- 函数原型

int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue));

- 参数说明
  - which：系统为每个进程提供 3 个时钟——ITIMER_REAL 设定绝对时间，经过指定的时间后，内核将发送 SIGALRM 信号给本进程；ITIMER_VIRTUAL 设定程序执行时间，经过指定的时间后，内核将发送 SIGVTALRM 信号给本进程；ITIMER_PROF 设定进程执行以及内核因本进程而消耗的时间和，经过指定的时间后，内核将发送 ITIMER_VIRTUAL 信号给本进程。
  - Value：设定的时间值，其定义如下：

```
struct itimerval {
        struct timeval it_interval;    /* next value */
        struct timeval it_value;       /* current value */
    };
    struct timeval {
        long tv_sec;                   /* seconds */
        long tv_usec;                  /* microseconds */
    };
```

- 返回值和错误代码

调用成功返回 0，否则返回-1。

（6）abort()函数异常退出信号。



● 功能

abort()函数将会导致进程的异常退出，除非该进程捕获了 SIGABRT 信号，并且该信号处理例程不返回。

● 引用的头文件

#include <stdlib.h>

● 函数原型

void abort(void);

● 返回值和错误代码

该函数无返回值。

8. 信号的安装函数

（1）signal()函数。

● 功能

signal()函数为由 signum 指定的信号安装新的信号处理例程。新的信号处理例程可以是用户指定的，也可以是 SIG_IGN 或者 SIG_DFL。

● 引用的头文件

#include <signal.h>

● 函数原型

void (*signal(int signum, void (*handler))(int)))(int);

● 参数说明

 - signum：要安装的信号编号。
 - Handler：指向新处理例程的函数指针，可以使用 SIG_IGN 代表忽略对该信号的处理，SIG_DFL 代表使用默认值处理该信号。

● 返回值和错误代码

调用成功，返回最后一次安装信号 signum 而调用 signal()函数时，传入的 handler 值；失败则返回 SIG_ERR。

（2）sigaction()函数。

● 功能

sigaction()函数用于改变进程对于收到一个特定信号后的行为，可以为除 SIGKILL 和 SIGSTOP 外的任何一个特定有效的信号。第二个参数是指向结构 sigaction 的一个实例的指针 act，在结构 sigaction 的实例中，指定了对特定信号的处理，如果为空，进程会以默认方式对信号处理；第三个参数 oldact 指向的对象用来保存原来对相应信号的处理，可指定 oldact 为 NULL。如果把第二、第三个参数都设为 NULL，那么该函数可用于检查信号的有效性。

● 引用的头文件

#include <signal.h>

● 函数原型

int sigaction(int signum,const struct sigaction *act,struct sigaction *oldact));

* 参数说明
  * signum：信号的编码。
  * act：该参数包含了对指定信号的处理动作、信号所传递的信息、信号处理函数执行过程中应屏蔽掉的函数等信息，其定义如下：

```
struct sigaction {
                union{
                    __sighandler_t _sa_handler;
                    void (*_sa_sigaction)(int,struct siginfo *, void *);
                }_u
                sigset_t sa_mask;
                unsigned long sa_flags;
                void (*sa_restorer)(void);
}
```

联合数据结构中的两个元素_sa_handler 以及*_sa_sigaction 用于指定信号关联函数，即用户指定的信号处理函数。除了可以是用户自定义的处理函数外，还可以为 SIG_DFL（采用默认的处理方式），也可以为 SIG_IGN（忽略信号）。

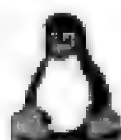
由_sa_handler 指定的处理函数只有一个参数，即信号值，所以信号不能传递除信号值之外的任何信息。而由_sa_sigaction 指定的信号处理函数带有 3 个参数，是专为实时信号而设计的（当然同样支持非实时信号），第一个参数为信号值，第三个参数没有使用（POSIX 标准没有规定使用该参数的标准），第二个参数是指向 siginfo_t 结构的指针，该结构中包含信号携带的数据值，参数所指向的结构如下：

```
typedef struct siginfo {
        int         si_signo;   /*信号值，对所有信号有意义*/
        int         si_errno;   /* errno 值，对所有信号有意义*/
        int         si_code;    /*信号产生的原因，对所有信号有意义*/

union {                   /*联合数据结构，不同成员适应不同信号*/
      int _pad[SI_PAD_SIZE];

      /*用于 kill()函数的定义*/
      struct {
            pid_t _pid;         /* sender's pid */
            uid_t _uid;         /* sender's uid */
      } _kill;

      /*用于 POSIX.1b timers 的定义*/
      struct {
            unsigned int _timer1;
```




```
            unsigned int _timer2;
        } _timer;

        /*用于 POSIX.1b signals 的定义*/
        struct {
            pid_t _pid;        /* sender's pid */
            uid_t _uid;        /* sender's uid */
            sigval_t _sigval;
        } _rt;

        /*用于 SIGCHLD 信号的定义*/
        struct {
            pid_t _pid;        /*which child */
            uid_t _uid;        /*sender's uid */
            int _status;       /*exit code */
            clock_t _utime;
            clock_t _stime;
        } _sigchld;

        /*用于 SIGILL, SIGFPE, SIGSEGV, SIGBUS 等信号的定义*/
        struct {
            void *_addr; /* faulting insn/memory ref. */
        } _sigfault;

        /*用于 SIGPOLL 信号的定义*/
        struct {
            int _band;  /* POLL_IN, POLL_OUT, POLL_MSG */
            int _fd;
        } _sigpoll;
    } _sifields;
} siginfo_t;
```

- sa_mask：指定了信号处理例程执行过程中，应当屏蔽的信号。此外，该信号本身也是默认屏蔽的，除非指定了 SA_NODEFER 或者 SA_NOMASK 标志。
- sa_flags：影响信号行为的标志。sa_flags 中包含了许多标志位，包括刚刚提到的 SA_NODEFER 及 SA_NOMASK 标志位。另一个比较重要的标志位是 SA_SIGINFO，当设定了该标志位时，表示信号附带的参数可以被传递到信号处理函数中，因此，应该为 sigaction 结构中的 sa_sigaction 指定处理函数，而不应该为 sa_handler 指定信号处理函数，否则，设置该标志变得毫无意义。即使为 sa_sigaction 指定了信号处理函数，如果不设置 SA_SIGINFO，信号处理函数同样不能得到信号传递过来的数据，在信号处理函数中对这些信息的

访问都将导致段错误（Segmentation fault）。

➢ sa_restorer：该参数已过时，POSIX 标准不支持它，不应再被使用。

- 返回值和错误代码

调用成功返回 0；否则，返回-1，并且将赋值 errno。

4.4.2 Linux 中的管道

原始的 UNIX 管道是利用有公共祖先的进程之间的共享文件描述符进行的一种通信方式，所有 UNIX 系统都支持这种通信机制。

管道是通过调用 pipe 系统调用而创建的，可以在使用 fork 创建的父子进程和兄弟进程间共享。pipe()函数的原型如下：

int pipe(int filedes[2]);

调用该函数后，在 filelds 参数中将返回两个文件描述符，这两个描述符分别连接管道的两端。filedes[0]代表管道的读端，是只读的；fileldes[1]代表管道的写端，是只写的。图 4-4 描述了这种连接方式。

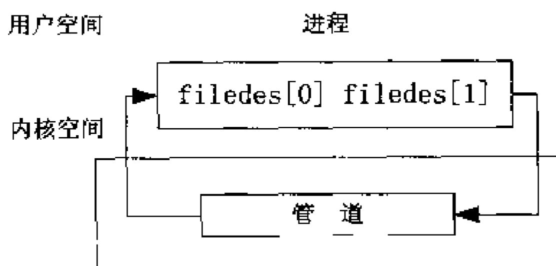


图 4-4 pipe 系统调用后管道连接方式

在这种方式中，管道可以看作是一种假想的容器，filedes[0]、filedes[1]分别代表容器的出口和入口。进程通过该容器通信，而访问该容器的互斥与同步由内核维护。

为了利用管道进行通信，在创建管道后，必须调用 fork()函数创建新的进程。第 3 章中已经提到，使用 fork()函数创建的父子进程间共享大量资源，包括文件描述符表。因此，pipe()函数返回的两个文件描述符也在父子进程之间共享。这样，父子进程（或兄弟进程）之间的管道连接就变成如图 4-5 所示的形式。

在确定了管道的传输方向后（父→子、子→父、兄→弟、弟→兄），关闭多余的文件描述符，读端关闭 filedes[1]，写端关闭 filedes[0]。图 4-5 可以简化成图 4-6 的形式。

到此为止，就成功地建立了一个进程间的单向 IPC 通道。

1. 管道的特点

由上面的叙述可以看出，管道通信具有以下特点：

- 管道是半双工的，数据只能向一个方向流动；需要双方互相通信时，需要建立起两个管道；只能用于实现父子进程或者兄弟进程之间（具有亲缘关系的进程）的



通信。

- 管道借用了文件描述符等来连接管道的两端，使得管道和文件有着相同的接口，但是管道并不是一种真正意义上的文件，它的存在是暂时性的。
- 管道是一种队列，数据按照先进先出的方式流经管道。写入的内容每次都添加在管道缓冲区的末尾，而每次都是从缓冲区的头部读出数据，并且读出的数据将被抛弃，不可恢复。
- 如果写一个读端已经关闭的管道，则会产生 SIGPIPE 信号，对该信号的默认处理将导致 write 出错返回，错误代码 error 设为 EPIPE。
- 管道能暂存的最大数据量由常数 PIPE_BUF 定义。

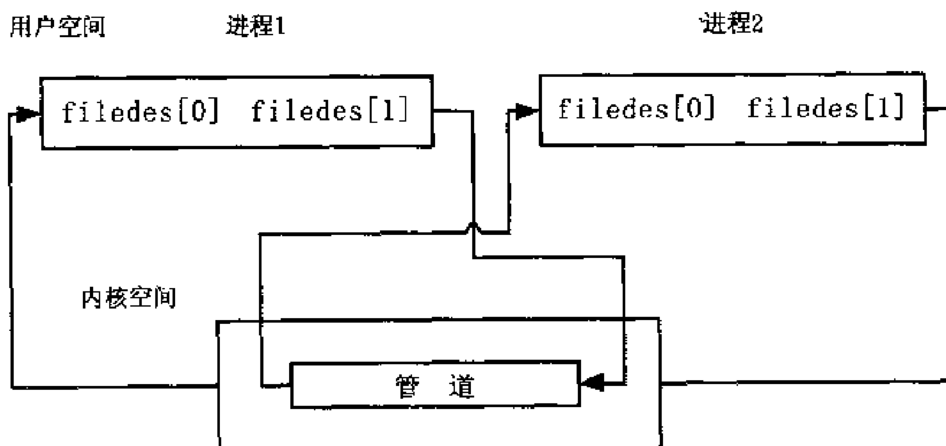


图 4-5 pipe 系统调用后父子进程间的管道链接方式

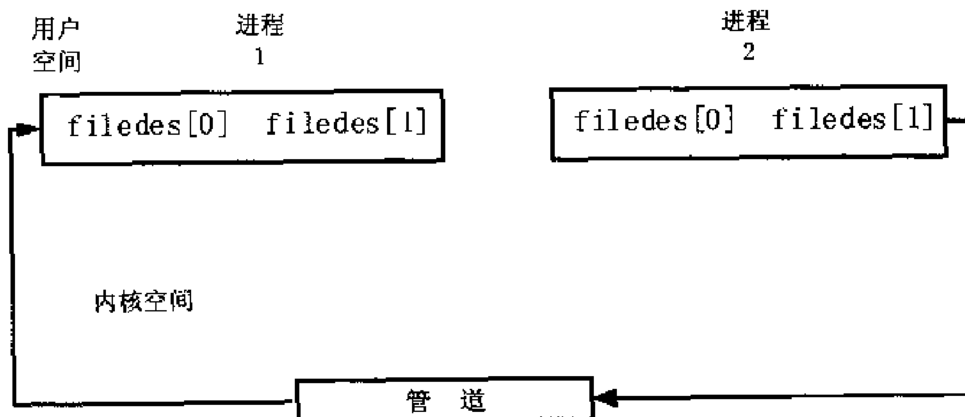


图 4-6 父子进程间的单向管道链接方式

2. 管道的系统调用

Linux 中管道的主要系统调用为 pipe()函数。

pipe()创建一个管道。

- 功能

pipe()函数创建一对指向新创建管道 i 节点的文件描述符，并将 filedes 指针指向它。filedes[0]用于读操作，filedes[1]用于写操作。

- 引用的头文件

#include <unistd.h>

- 函数原型

int pipe (int filedes[2]);

- 参数说明
  - fieldes[0]：指向管道的读端。
  - fieldes[1]：指向管道的写端。
- 返回值和错误代码

调用成功返回 0；否则，返回-1，并且将赋值 errno。

4.4.3 先进先出文件 FIFO

FIFO 又称为命名管道。之所以被称为命名管道，是由于 FIFO 在文件系统的基础上为管道操作提供了命名服务。FIFO 通过一个真实的文件名来访问，这使得多个没有“血缘关系”的进程可以通过 FIFO 进行通信。相对于 FIFO，原始 UNIX 管道又被叫做匿名管道，它只在操作期间存在于内存中，而 FIFO 可以永久地存储在硬盘之上。

为了创建 FIFO 文件，可以在 Shell 提示符下使用 mkfifo 命令或者在程序中使用 mkfifo 系统调用。

在 Shell 提示符下使用 mkfifo 命令创建 FIFO 很容易，其命令如下所示：

mkfiflo [OPTION] NAME

这个命令将创建一个称为 NAME 的先进先出文件，可以使用-m 选项指定 FIFO 文件的访问权限，这与 chmod 命令是一样的。比如，下面的命令将创建一个所有用户都有读写权限的 FIFO 文件，其文件名叫做 myfifo。

[root@zzcLinux root]#mkfifo –m666 myfifo

用户也可以通过调用而 mkfifo 系统调用来创建 FIFO。其原型为：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo ( const char *pathname, mode_t mode );
```

其中，pathname 表示被创建的文件名称；mode 表示将在该文件上设置的权限位。一旦先进先出 FIFO 文件已经被创建，它就可以利用标准的文件 I/O 操作来访问。

1. FIFO 的操作

当操作 FIFO 时，应该注意以下几点：

- 可以使用标准的文件 I/O 函数（open,close,read,write,unlink 等）操作 FIFO。



- 当用 Open()调用打开 FIFO 后，一个 FIFO 和一个匿名管道具有同样的功能。当管道为空的时候，read()函数被阻塞；当管道是满的时候，write()函数被阻塞，且这时用 fcntl()函数设置 O_NONBLOCK 标志时，将引起 read()和 write()函数立即返回。
- 若写一个尚无进程为读而打开的 FIFO 时，则产生信号 SIGPIPE，若某个 FIFO 的最后一个写进程关闭了该 FIFO，则将为该 FIFO 的读进程产生一个文件结束标志。
- FIFO 有多个读进程和/或多个写进程的情况，在应用中是有用的。比如多个客户进程通过一个 FIFO 向服务器进程传送信息。在这种情况下，需要有一个规则来决定如果多个写进程企图同时写这个 FIFO 该如何处理。这个规则是：一个 write()函数可以写管道能容纳（Linux 下为 4KB）的任意个字节，这种写操作保证是互斥进行的。多个写操作的数据在 FIFO 文件中相互分离。

2. FIFO 的用途

FIFO 管道通常用于以下两种情况：

- FIFO 由 Shell 命令使用，以便将某个程序的输出作为输入送往另一程序。这样，可以避免创建中间临时文件。
- FIFO 用于客户—服务器应用程序中，作为它们之间传递数据的通道。

3. FIFO 的系统调用

Linux 中 FIFO 的主要系统调用有：mkfifo ()。

使用 mkfifo 创建一个管道。

- 功能

mkfifo 使用 pathname 创建一个特殊的 FIFO 文件，特殊的 FIFO 文件类似于管道，它利用文件系统，提供了一种命名管道机制。FIFO 文件创建后，所有进程都可以像操作普通文件一样操作它。在默认情况下，读一个写端尚未打开的 FIFO 文件时，读进程会被阻塞。

- 引用的头文件

```
#include <sys/types.h>
#include <sys/stat.h>
```

- 函数原型

```
int mkfifo (const char *pathname,mode_t mode);
```

- 参数说明
  - pathname：创建的 FIFO 文件路径名。
  - mode：指定该 FIFO 文件的操作权限。
- 返回值和错误代码

调用成功返回 0；否则，返回-1，并且将赋值 errno。

4.4.4 System V IPC 机制

在 UNIX 的 System V 中，AT&T 加入了 3 种新形式的 IPC 机制：共享存储区（Shared

Memory)、消息队列(Message Queue)和信号量(Semaphore)。消息允许进程发送格式化的数据流到任意的进程;共享存储区允许进程间共享它们虚地址空间中的部分区域;信号量允许进程间同步的执行。这 3 种机制间有很多共有的性质,比如标识方法、权限管理和类似的接口等。

为了对这 3 种机制进行统一管理,SystemV 引入了一些新的概念。

1. IPC 标识符

内核中的每个 IPC 对象(共享存储区、消息队列和信号量)都用一个非负整数来标识,通常把这个整数叫做 IPC 标识符。对该 IPC 对象的操作,都必须通过 IPC 标识符进行。例如,对一个消息队列发送或读取消息时,只需要知道其队列标识符。

不同种类的 IPC 对象的标识符是独立编号的,即系统中可以同时存在一个共享存储区、消息队列和信号量的实例,其 IPC 标识符都是 12345。

2. IPC 关键字

在创建任何一种 IPC 对象时,都需要指定一个关键字(key),关键字被定义为 key_t 类型。通常在头文件<sys/types.h>中,key_t 被定义为长整型。关键字 IPC_PRIVATE 被用于创建一个新的 IPC 对象。

3. IPC 的权限管理结构

Linux 中,每一个 IPC 对象都有一个 ipc_perm 结构。该结构规定了 IPC 对象的许可权和所有者。该结构的定义如下(在 linux\linux2.4.x\include\linux\Ipc.h 中):

```
struct ipc_perm {
    key_t   key;
    ushort uid;        /*owner euid and egid */
    ushort gid;
    ushort cuid;       /* creator euid and egid */
    ushort cgid;
    ushort mode;       /* lower 9 bits of shmflg */
    ushort seq;        /* sequence number */
};
```

其中,比较重要的字段有:key 是该 IPC 对象的关键字;uid、gid 代表所有者的有效用户标识号和有效组标识号;cuid、cgid 代表创建者的有效用户标识号和有效组标识号;mode 字段规定了对象的操作权限。

在创建 IPC 对象时,ipc_perm 结构中除 seq 以外的字段都将被赋初值。以后,可以调用 IPC 对象的控制函数 msgctl()、semctl()或 shmctl()来修改 uid、gid、cuid、cgid 和 mode 字段。为了改变这些值,调用进程必须是 IPC 对象的创建者或超级用户,而更改这些字段的方法类似于改变文件的权限位的方法。



4. IPC 对象的控制

System V 的 IPC 对象是在系统范围内起作用的。因此，某个程序创建的 IPC 对象的生命期可以超出该程序的生命期。为此，Linux 系统提供了专门的命令，用于操作 IPC 对象。

5. IPC 对象的创建

3 种 IPC 的对象的 get 函数（msgget、semget 和 shmget）都有两个类似的参数 key 和 flag。当这两个参数满足下列条件之一时，get 函数则创建一个新的 IPC 结构（通常该操作由服务器程序进行）：

- key 是 IPC_PRIVATE。
- key 的值未与当前特定类型的 IPC 结构相匹配，并且在 flag 中指定了 IPC_CREAT 位。

如果 key 与系统中已有的 IPC 结构的 key 相同时，get 函数将返回现存的对象。如果要确保创建一个新的 IPC 对象，而不是引用具有同一标识符的已存在 IPC 对象，可以在 flag 中同时指定 IPC_CREAT 和 IPC_EXCL 位。这样，如果指定 key 的 IPC 对象已存在，get 函数将返回 EEXIST。

6. ipcs 命令

命令 ipcs 用于查询 System V IPC 对象的信息。其格式如下：

```
ipcs  [  -asmq  ]  [  -tclup  ]
ipcs  [  -smq   ] –i   id
ipcs  -h
```

该命令的使用方法如下：

- -I：查询 IPC 对象的 ID。
- -q：只显示消息队列对象。
- -s：只显示信号量对象。
- -m：只显示共享内存对象。
- -h：显示帮助信息。

默认情况下，此命令将同时显示 3 种 IPC 目标。下面是一个 ipcs 命令的应用实例。

```
[root@zzcLinux root]#ipcs
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems      status

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

7. ipcrm 命令

命令 ipcrm 用于将 IPC 对象从系统内核中删除。虽然也可以通过系统调用在程序中删除 IPC 对象，但在一般情况下，特别是在开发环境中，经常需要手工删除 IPC 对象。其格式如下：

ipcrm <msg | sem | shm> <IPC ID>

该命令中，需要指出删除对象的类型（msg、sem、shm）和 ID，下面是一个 ipcrm 使用的例子。

[root@zzcLinux root]#ipcrm shm 1

8. 消息队列

消息队列本质上是一种报文传递系统，发送到消息队列中的消息将被链接到一个内核链表中，随后接收进程将从该链表中取走消息。Linux 系统中，有 4 个系统调用可用于消息队列管理：msgget 用于返回（有可能是创建）一个消息描述符，该描述符将指定一个消息队列以便用于其他系统调用；msgctl 可设置和返回与一个消息描述符相关联的参数的选项，以及用于删除消息描述符的选项；msgsnd 用于发送一条消息；msgrcv 用于接收一条消息。

下面分别介绍这 4 个系统调用的使用方法。

（1）msgget 取得一个消息队列标识符。

- 功能

msgget 用于返回与参数 key 相关联的消息队列的标识符。如果参数 key 的值为 IPC_PRIVATE 或者 key 所指定的消息队列尚未创建，并且 msgflag 中指定了 IPC_CREAT，则创建一个新的消息队列。当同时指定 IPC_CREAT 和 IPC_EXCL 时，如果 key 指定的消息队列已存在，则该函数失败。

- 引用的头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

- 函数原型

```
int  msgget  ( key_t key,int msgflg);
```

- 参数说明
  - key：消息队列关键字。
  - msgflag：创建参数：

    IPC_CREAT：如果内核中没有此队列，则创建它。

    IPC_EXCL：与 IPC_CREAT 一起使用时，如果队列已经存在，则失败。

- 返回值和错误代码

如果调用成功，返回消息队列的标识符，如果失败，则返回-1。

（2）msgsnd 发送一条消息。

- 功能



发送一条消息到 msgid 指定的消息队列。如果消息队列已满，那么此消息不会写入到消息队列中，控制将返回到调用进程。默认情况下，调用进程将会挂起，直到消息队列中有空闲空间可以写入为止。

- 引用的头文件

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

- 函数原型

int msgsnd (int msgid,struct msgbuf *msgp,size_t msgsz,int msgflg);

- 参数说明
  - ➢ msgid：消息队列标识符，由系统调用 msgget 返回。
  - ➢ msgp：指向消息缓冲区的指针。
  - ➢ msgsz：消息的大小（以字节计算）。
  - ➢ msgflg：发送标志，可以设置为 0，表示忽略该参数。或者使用 IPC_NOWAIT 表示消息队列满时，不等待，立即返回。

- 返回值和错误代码

如果成功，返回 0；如果失败，则返回-1。

（3）msgrcv 接收一条消息。

- 功能

msgrcv 用于从指定的消息队列中接收一条消息。消息接收后，将从消息队列中删除该消息。如果当前消息队列中没有满足 msgrcv 参数要求的消息，默认情况下，调用进程将被挂起，直到队列中有一条满足条件的消息到来。如果在调用中指定了 IPC_NOWAIT 标志，在此情况下，msgrcv 将返回 ENOMSG。

- 引用的头文件

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

- 函数原型

int msgrcv (intmsgid, struct msgbuf *msgp, size_t msgsz,long mtype, int msgflg);

- 参数说明
  - ➢ msgid：消息队列标识符，由系统调用 msgget 返回。
  - ➢ msgp：指向接收消息的缓冲区的指针。
  - ➢ msgsz：缓冲区的大小（以字节计算），不包括 mtype 的长度。
  - ➢ mtype：要接收的消息的类型。如果参数为 0，将接收队列中最先到达的一条消息。
  - ➢ msgflg：发送标志，可以设置为 0，表示忽略该参数；或者使用 IPC_NOWAI，表示消息队列满时，不等待，立即返回。如果 msgflg 中指定了 MSG_NOERROR 标志，且消息的实际长度大于 msgsz，那么消息将会被截断，只返回 msgsz 长度的消息内容，并且内核将删除整条消息。

- 返回值和错误代码

如果成功，返回复制到消息缓冲区的字节数；如果失败，返回-1。

（4）msgctl 消息队列控制函数。

- 功能

该函数用于在 msgid 指定的消息队列上，执行由 cmd 指定的操作。该函数允许用户接收消息队列的信息，设置拥有者和组拥有者，修改共享存储区的权限。

- 引用的头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

- 函数原型

```
int msgctl ( int msqid,int cmd ,struct msqid_ds * buf ) ;
```

- 参数说明
  - msqid：消息队列 ID。
  - cmd：当前有效的命令值有下面几个：

    IPC_STAT：读取消息队列的数据结构 msgid_ds，并将其存储在 buf 指定的地址中。

    IPC_SET：设置消息队列的数据结构 msgid_ds 中的 ipc_perm 元素的值。

    IPC_RMID：从系统中删除消息队列。

    Buf：命令的输出输入缓冲区。

- 返回值和错误代码

调用成功返回 0，失败返回-1。

9. 共享存储器

进程通过共享它们虚拟地址空间的若干部分，然后对存储在共享存储器中的数据进行读和写来实现彼此间的直接的通信。共享内存可以说是最快的 IPC 机制，因为数据不需要在进程间复制。使用共享内存需要注意的一点是，共享存储区操作之间同步应该由用户自己来维护，比如在一个进程的写操作完成之前，另一个进程不能读取相同的地址。通常情况下，可以使用后面提到的信号量的实现同步访问共享存储区。

操作共享存储区的系统的调用类似于对消息队列的系统调用。shmget 建立一个新的共享存储区或是返回一个已存在的共享存储区；shmat 从逻辑上将一个共享存储区链接到一个进程的虚拟空间上；shmdt 断开一个共享存储区和进程的链接；shmctl 对与共享存储区相关联的各种参数进行操作。

当一个共享存储区链接到进程的地址空间后，进程可以像访问普通存储区一样地读写共享存储区，不再需要附加的系统调用来存取共享存储区中的数据。

下面详细介绍上面提到的几个系统调用。

（1）shmget 取得一个共享储存区的标识符。

- 功能



shmget 用于返回与参数 key 相关联的共享存储区的标识符。如果参数 key 的值为 IPC_PRIVATE 或者 key 所指定的共享存储区尚未创建，并且 shmflag 中指定了 IPC_CREAT，则创建一个新的消息队列。当同时指定 IPC_CREAT 和 IPC_EXCL 时，如果 key 指定的共享存储区已存在，则该函数调用失败。

- 引用的头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

- 函数原型

```
int #shmget（key_t key,int size,  int shmflg）
```

- 参数说明
  - key：共享存储区关键字。
  - Size：创建的共享存储区的大小。
  - Shmflag：创建参数：

    IPC_CREAT：如果内核中没有此队列，则创建它。

    IPC_EXCL：与 IPC_CREAT 一起使用时，如果队列已经存在，则失败。
- 返回值和错误代码

如果成功，将返回共享存储区的标识符；如果失败，返回-1。

（2）shmat 链接共享存储区到进程空间。

- 功能

shmat 用于将 shmid 指定的共享存储区链接到调用进程的数据段地址空间。链接到的地址由 shmaddr 指定，shmaddr 可以有以下几种取值方式：

  - 如果 shmaddr 为 0，则此段链接地址由内核决定。
  - 如果 shmaddr 为非 0，并且没有在 shmflg 中指定 SHM_RND，则此段链接到 shmaddr 指定的地址上。
  - 如果 shmaddr 为非 0，并且在 shmflg 中指定了 SHM_RND，则此段链接到（shmaddr-(shmaddr mod SHMLBA)）所表示的地址上。其中，SHM_RND 代表取整操作，SHMBA 代表向下边界对齐。

通常情况下，不用指定共享存储区的链接位置，可以将 shmaddr 定为 0，由内核选择地址。

- 引用的头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

- 函数原型

```
int shmat（int shmid,const char *shmaddr,int shmflg）
```

- 参数说明
  - shmid：共享存储区的 ID。

- ➢ Shmaddr：共享存储区链接的地址。
- ➢ Shmflg：共享存储区标志：

  SHM_RDONLY：指定该标志，将以只读方式链接该段。否则，以读写方式链接该段。

  SHM_RND：链接的地址取整。

- ● 返回值和错误代码

调用成功，返回该段所链接的实际地址，如果出错，返回-1。

（3）shmdt 断开共享存储区与进程的链接。

- ● 功能

shmdt 用于从调用进程的数据段中断开位置处于 shmaddr 的共享存储区。要断开的共享存储区必须是当前已经链接的进程的地址空间，shmaddr 必须是调用 shmat 返回的值。

- ● 引用的头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

- ● 函数原型

```
int shmdt (const char *shmaddr)
```

- ● 参数说明

shmaddr：共享存储区链接的地址。

- ● 返回值和错误代码

如果失败，返回-1。

（4）shmctl 共享存储区控制函数。

- ● 功能

shmctl 用于在 msgid 指定的共享存储区上，执行由 cmd 指定的操作。该函数允许用户接收共享存储区的信息，设置拥有者和组拥有者，修改共享存储区的权限。该函数也用于删除一个共享存储区对象。

- ● 引用的头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

- ● 函数原型

```
int shmctl (int shmid, int cmd, struct shmid_ds *buf)
```

- ● 参数说明
  - ➢ shmid：共享存储区 ID。
  - ➢ Cmd：当前有效的命令值有下面几个：

    IPC_STAT：读取共享存储区的数据结构 shmid_ds，并将其存储在 buf 指定的地址中。

    IPC_SET：设置消息队列的数据结构 shmid_ds 中的 ipc_perm 元素的值。



IPC_RMID：该命令并不真的删除该共享存储区，而是将其标记为可删除，真正的删除操作在最后一个 shmdt()系统调用后进行。

➢ Buf：命令的输出输入缓冲区。

● 返回值和错误代码

调用成功返回 0，调用失败返回-1。

10. 信号量

信号量用于多个进程的同步与互斥操作。在信号量实现以前，如果一个进程要锁定一个资源，必须使用文件锁来进行。使用文件锁有一个缺点，就是锁定文件，往往因为同名文件已经存在而失败，导致系统误认为已有一个进程锁定了该资源。

在 4.2 节中，已经详细介绍了信号量的原理。而 Linux 系统实现了基于信号量集的信号量机制。下面，逐一介绍信号量集操作的系统调用。

（1）semget 取得一个信号量集的标识符。

● 功能

semget 用于返回与参数 key 相关联的信号量集的标识符。如果参数 key 的值为 IPC_PRIVATE 或者 key 所指定的信号量集尚未创建，并且 semflag 中指定了 IPC_CREAT，则创建一个新的信号量集。当同时指定 IPC_CREAT 和 IPC_EXCL 时，如果 key 指定的信号量集已存在，则该函数调用失败。

● 引用的头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

● 函数原型

```
int semget (key_t key, int nsems, int semflg);
```

● 参数说明

➢ key：信号量集关键字。

➢ nsems：如果该函数用于创建一个新的信号量集，则 nsems 指定要创建的信号量的个数。

➢ semflag：信号量创建参数：

◆ IPC_CREAT：如果内核中没有此信号量集，则创建它。

◆ IPC_EXCL：与 IPC_CREAT 一起使用时，如果队列已经存在，则调用失败。

● 返回值和错误代码

如果调用成功，则返回信号量集的标识符；如果调用失败，则返回-1。

（2）semop 信号量操作函数。

● 功能

semop()函数可以对 semid 指定的信号量集的指定成员进行操作。

sops 所指向的数组中的一个元素（共 nsops 个）就表示了对某个信号量的一次操作。该系统调用保证这些操作的原子性。即该数组指定的所有操作，要么全部执行，要么都不

执行。这也就是前面谈到的信号量集合的概念。

- 引用的头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

- 函数原型

```
int semop ( int semid, struct sembuf *sops,unsigned nsops)
```

- 参数说明
  - semid：信号量 ID。
  - sembuf：指向将要执行的操作数组的指针。其定义如下（在 linux\linux2.4.x/include/linux/sem.h 中）：

```
struct sembuf {
    unsigned short  sem_num;  /* semaphore index in array */
    short           sem_op;   /* semaphore operation */
    short           sem_flg;  /* operation flags */
};
```

其中，sem_num 表示信号量在信号量集中的编号（第一个信号量的编号是 0，然后依次递增）；sem_op 是一个短整型，表示对信号量的操作，正数代表释放资源，负数代表减少资源数；sem_op 为 0，则表示该进程将睡眠，直到信号量的值为 0；sem_flg 表示操作的标志，当前支持的标志有 IPC_NOWAIT 和 SEM_UNDO。如果指定了 IPC_NOWAIT，那么当参数 sem_op 为负，要求申请占用资源时，如果条件不满足，进程立即返回，并不等待。如果指定了 SEM_UNDO，则进程退出时，将进行信号量的清理操作。

  - nsops：sops 指向的数组的元素个数。

- 返回值和错误代码

调用成功返回 0，调用失败返回-1。

（3）semctl 信号量控制函数。

- 功能

semctl()用于在 semid 指定的信号量集（或者是该信号量集中的第 semnum 个信号量）上执行由 cmd 指定的操作。该函数允许用户接收信号量集的信息，设置拥有者和组拥有者，修改信号量集的权限。该函数也用于删除一个信号量集对象，删除一个信号量集时，将唤醒所有在该信号量集上等待的任务。

- 引用的头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

- 函数原型

```
int semctl (int semid, int semnum, int cmd, union semun arg)
```

- 参数说明



- semid：信号量集 ID。
- cmd：当前有效的命令值有下面几个：
  - IPC_STAT：读取信号量集的数据结构 semid_ds，并将其存储在 arg.buf 指定的地址中。
  - IPC_SET：设置信号量集的数据结构 semid_ds 中的 ipc_perm 元素的值。
  - IPC_RMID：该命令立即删除该信号量集，并将唤醒所有在该信号量集上等待的任务。这些任务的 semop 操作失败返回，其错误代码 error 被设为 EIDRM。使用该命令，调用进程必须具有超级用户权限，或是信号量集的创建者或拥有者。
  - GETALL：读取信号量集中的所有信号量的值，并存放到 arg.array 中。
  - GETNCNT：返回正在等待的进程数目。
  - GETPID：返回最后一个执行 semop 操作的进程的 PID。
  - GETVAL：返回信号量集中的第 semnum 个信号量的值。
  - GETZCNT：返回该信号量集上等待完全空闲的资源的进程数。
  - SETALL：使用 arg.array 中的值设置信号量集中所有信号量，如果某个值大于零，则唤醒等待该信号量的任务。
  - SETVAL：使用 arg.val 中的值设置信号量集中第 semnum 个信号量的值 arg。

- 返回值和错误代码

调用成功，返回一个正数；失败，返回-1。

4.5 小结

操作系统必须提供任务同步与通信机制，以解决并发执行的多道程序之间的协调问题。使用信号量机制来保护任务中的关键临界区，使得任务能够以互斥的方式访问资源，但是在实现互斥的同时，也必须考虑到死锁的可能性。

传统的 UNIX 系统，使用管道机制和信号机制进行通信，而基于 System V 的 IPC 机制（消息队列、共享存储区、信号量）则提供了新的高效的任务间通信方法。

4.6 思考题

1. 并发多个任务间可能有哪几种关系？应该如何处理？
2. 同步机制应遵循哪些基本原则？为什么？
3. 死锁问题有哪些解决办法？各自的优缺点是什么？
4. 优先级反转产生的原因和解决办法是什么？
5. 任务间通信的常用方法及其优缺点是什么？
6. 请思考将半双工管道改造为全双工管道的办法。

第5章 存储器管理

知识点：

- 存储器管理的基本概念
- 连续存储管理方法及其优缺点
- 分段、分页管理
- 虚拟存储管理的原理与实现
- Linux 的存储器管理

本章导读：

存储器是现代操作系统进行操作的中心，是计算机系统中一种数量有限的重要资源。操作系统作为系统资源管理者，必须对存储器加以高效的管理。当今个人计算机的存储容量已是1960年IBM 7094大型计算机存储容量的10倍，然而程序的大小也正好增长10倍，程序的增大正好填满增大的存储器。因此，如何有效地管理存储器仍然是一个值得探讨和解决的问题。



5.1 存储器管理概述

存储器是计算机系统中必须认真管理的重要资源。近年来，随着计算机硬件技术的发展，存储器的容量一直在不断扩大。但是，相对于应用程序的增长速度而言，存储器硬件技术的发展仍然严重滞后。16 位地址总线的计算机只能寻址 64KB 的地址空间；20 位地址总线的计算机可以寻址 1MB 的地址空间；而目前的主流机型 32 位地址总线的计算机已经支持 4GB 的地址空间，下一代的 64 位地址总线的计算机支持的地址空间将高达 2^{64} 字节。因此，在目前来说，如何有效地管理存储器资源仍然是一个很重要的问题。

首先需要澄清的是，并不是所有的计算机系统都需要存储器管理功能。在早期的计算机系统中，没有多道程序的概念，整个系统中只有一道程序在运行。该程序可以使用系统的所有资源，包括全部的存储器。这种一道程序独占整个计算机系统的情况，在现在的一些嵌入式设备中仍然存在。对这种系统来说，存储器管理相当简单，甚至可以说是没有。

存储器的体系结构和管理方法是计算机系统中演化得非常快的部分。从原始的纸带存储到磁芯存储，再到各种大容量的 RAM 芯片，存储管理的硬件介质容量不断扩大；从连续静态固定分区管理到基于分段、分页的内存管理，再到基于虚拟存储器的段页式管理，存储器的管理方法也在不断发展；从无 Cache 系统到微处理器芯片内集成 Cahce，到板载一级 Cache、二级 Cache 甚至三级 Cache，存储器系统的结构也在不断演化中。但是，就存储器管理的目的来看，现代存储器管理主要有以下几个目标：

- 为多道程序设计提供支持。
- 提高内存利用率。
- 简化内存的使用，为用户开发应用程序提供支持。

引入多道程序设计的主要目的是提高微处理器的利用率。而为了实现多道程序的并行运行，存储器管理系统必须能够将内存分为多个部分，每部分都装入一道程序，以便多道程序的并行运行。而现代的微处理器一般都提供了相当大的地址空间（比如 4GB），而实际物理内存容量一般达不到这个级别，因此，为了支持这么大的地址空间，存储器管理必须提供虚拟存储器管理能力。即在大容量外存的支持下，通过将部分不立即使用的内存交换到外存上，以满足正在运行的进程对超过实际物理内存大小的存储器的请求。

影响内存利用率的一个重要因素是内存的管理开销。一般来说，为了便于管理（分配和回收），内存分配有一个最小单位。小于该单位的内存请求，将按此最小单位分配内存，其多余的未使用空间被称为“碎片”。减小碎片的方法，是缩小该基本单位。从内存的分区管理到分段管理，再到分页管理和分段分页结合的段页式管理都体现了这一思想。

另外，为了方便用户使用，存储器管理应该提供相应的功能，以支持用户的应用程序开发。比如分段机制可以帮助用户按功能组织代码，共享存储器机制可以方便用户高速通信。下面将逐一介绍这些概念和内容。

5.1.1 基本概念

1. 地址转换

现代操作系统将存储器管理分为两个层次：内存和外存。内存是指计算机配备的物理内存。物理内存的地址是计算机内存的真实地址，也称绝对地址。外存可以是硬盘等大容量随机访问设备。

为了提高物理内存的利用率，可以采用分片的形式，即把内存空间分成大小不等的区域或大小相等的小块，这就意味着要解决内存区域如何分配、各区域内的信息如何保护等问题。直接把物理地址提交给用户使用是不现实的，因为这样将增大用户开发程序的难度，并且用户各自管理内存难免会产生冲突。而且，在采用了多道程序设计技术之后，多个用户程序共享内存，由用户程序自行分配内存更是不可能的事。

为了支持多道程序设计，方便用户使用，操作系统往往为单个任务提供一个相对独立的地址空间。这些地址空间都从 0 地址开始，到最高地址结束。这种地址空间称为逻辑地址空间，也可称为虚拟空间。而虚地址到实际物理内存地址的映射由操作系统维护。使用这种机制，可以使用户摆脱繁琐的内存管理。

2. 程序的装入

在多任务环境下，程序要运行必须为之创建新任务。通常情况下，由装入程序（bootloader）读入该程序的数据和代码放置到合适的内存位置。该过程中，装入程序需要把该程序的逻辑地址转换成内存中的物理地址，该操作叫做地址变换或称地址映射。在多数情况下，地址变换过程都要进行地址的重新定位。重定位又可分为静态重定位和动态重定位两种方式。

- 静态重定位

静态重定位是在程序执行前就进行地址变换的方式。在程序装入内存开始运行后，直到程序运行结束，地址不再变动，这个工作往往是由重定位装入程序来完成的。这种重定位方式不需要硬件支持，实现起来比较简单，但程序一旦进行了重定位就再也不能移动也不能重新分配内存，所以不利于内存空间的有效利用。

- 动态重定位

动态重定位是在程序执行过程中要访问指令或数据时才进行地址变换的方式。这种方式中，代码中的地址都是相对地址。在访问该地址时，需要将该地址与基准地址相加，得到真实的绝对地址后才能进行。为了提供地址转换的速度，动态重定位方式通常需要硬件的支持。

通常，MMU（内存管理单元）中专门设置了用于地址定位的重定位寄存器，当存储管理系统为作业分配了一个内存区域后，就把该区的起始地址放到重定位寄存器中。这样一来，程序使用的地址将是它的逻辑地址和重定位寄存器的内容之和。

一个程序装入时，指令的相对地址加上定位寄存器的值就可以得到它的物理地址。程



序执行时，操作数的地址也通过这种方式计算出来。由于动态重定位方式允许在指令执行过程中实现操作数的地址转换，因而，采用动态重定位方式可使得用户进程在内存中自始至终保持其原有的逻辑空间关系。这样，系统只要改变重定位寄存器的内容，就可以将程序的地址空间改变成另一个地址空间，为程序在内存中浮动提供了方便的手段。所以它有利于共享，对内存的使用较为方便灵活，而且更为有效。但这种方式增加了硬件支持，在实现动态重定位时存储管理的软件也较为复杂。

3. 覆盖

覆盖技术与交换技术是在多道环境下扩充内存的两种方法，用以解决在较小的存储空间中运行大型程序时遇到的矛盾。

覆盖是指一个任务的若干程序段，或多个作业的某些部分共享某一个存储空间。覆盖技术的实现是把程序划分为若干个功能上相对独立的程序段，按照其自身的逻辑结构使那些不会同时执行的程序段共享同一块内存区域。程序段先保存在磁盘上，当有关程序段的前一部分执行结束后，再把后续程序段调入内存，覆盖前面的程序段。

覆盖不需要任何来自操作系统的特殊支持，可以完全由用户实现，即覆盖技术是用户程序自己附加的控制。覆盖技术要求程序员提供一个清楚的覆盖结构，即程序员只要把一个程序划分成不同的程序段，并规定好它们的执行和覆盖的顺序，操作系统即根据程序员提供的覆盖结构完成程序段之间的覆盖。

覆盖可以由编译程序提供支持：被覆盖的块是由程序员或编译程序预先确定的。

4. 交换

在多道程序环境下，可能存在占用内存的任务因为等待某事件的发生而阻塞，而其他任务却因为内存不足而没法装载运行。显然，这对系统资源是一种严重的浪费，并且降低了系统的吞吐量。为了解决这一问题，在系统中引入了交换的概念。所谓“交换”是指把内存中暂时不能运行的进程或暂时不用的代码和数据换出到外存上，以便腾出足够的内存空间，把具备运行条件的任务或是任务所需要的数据和代码换入内存。合理的交换可以有效地提高内存的利用率和系统的吞吐量，交换技术现已被广泛地应用在各种操作系统中。

5. 内存共享

所谓内存共享是指两个或多个进程共用内存中相同区域，这样不仅能使多道程序动态地共享内存，提高内存利用率，而且还能共享内存中某个区域的信息。共享的内容包括：代码共享和数据共享，特别是代码共享要求代码必须是纯代码。

内存共享的一个目的是通过代码共享节省内存空间，提高内存利用率；另一个目的是通过数据共享实现进程通信。

6. 地址空间保护

通常，内存中的程序有多个，不仅有用户任务，而且还有操作系统。特别是多道程序运行环境下的用户任务可能不止一个，为避免内存中多个任务间相互干扰，必须对内存采取保护措施，以保证各道程序都在自己所属的空间中或在公共区域中工作。存储保护通常

需要有硬件支持，并由软件配合实现。

存储保护的内容包括：保护系统程序区不被用户有意或无意地侵犯；不允许用户程序读写不属于自己地址空间的数据，如系统区地址空间、其他用户程序的地址空间。

常见的内存保护有两种：防止地址越界和防止操作越权。

- 防止地址越界

每个任务都具有其相对独立的地址空间，如果任务在运行时所产生的地址超出其地址空间，则发生地址越界。地址越界可能侵犯其他任务的空间，从而影响其他任务的正常运行；也可能侵犯操作系统空间，从而导致系统混乱。因此，对进程所访问的地址必须检查，防止其越界操作。

- 防止操作越权

对于允许多个进程共享的公共区域，每个进程都有自己的访问权限。例如，有些进程可以执行写操作，而其他进程只能执行读操作等。因此，必须对公共区域的访问加以限制和检查。下面列出了操作共享地址空间的一般规则：

- 对属于自己区域的信息，可读可写。
- 对公共区域中允许只读共享的信息，只可读而不可修改。
- 对公共区域中允许读写共享的信息，可读可写。

7. 内存容量的“扩充”和虚拟存储器

理想情况下，用户在编写程序时，不应该受到内存容量的限制，所以大多数通用操作系统都采用一定技术来“扩充”内存的容量，使用户得到比实际内存容量大得多的内存空间。

内存容量“扩充”的基本原理如下：软件、硬件相互协作，将内存、外存结合起来统一使用。利用程序局部性的原理，通过借助虚拟存储技术或其他交换技术，达到在逻辑上扩充内存容量的效果，即为用户提供比内存物理空间大得多的地址空间，使得用户感觉它的作业是在这样一个大的存储器中运行。

5.1.2 内存管理的分类

内存管理有多种分类方法，按照内存分配是否必须连续进行，内存管理可以分为连续分配方式和离散分配方式。早期的分区内存管理属于连续内存分配，而目前的分页、分段等内存管理则属于离散内存分配方式。按照内存分配是否支持超过真实物理内存的大小的地址空间，内存管理可以分为实内存管理和虚内存管理等。

5.1.3 早期连续内存分配

早期内存管理，大多基于连续内存的分区分配。分区管理是能满足多道程序运行的最简单的存储管理方案。其基本思想是把内存划分成若干个连续区域，每个分区装入一个运



行程序。分区的方式可以归纳成固定分区和可变分区两类。

1. 固定式分区

固定式分区的基本思想是在系统生成时就将主存划分为若干个分区，每个分区的大小可以不等，但事先必须固定，以后也不能改变。图 5-1 所示是固定分区的一个示例。操作系统空间（40KB）以后的内存按照大小递增的方式（8KB、32KB、64KB、112KB）分成了 4 个区。固定式分区在任务未装入时，分区的大小、数目已预先确定，这样容易造成分区内的碎片问题，从而影响内存的利用率。为此，引入了可变式分区的概念。

2. 可变式分区

可变式分区也就是动态划分存储器的分区方法，它是在作业装入和处理过程中建立的分区，并且要使分区的容量正好能适应作业的大小。在任务进入系统前，将根据任务的大小来申请所需存储容量，然后由系统实施分配。图 5-2 所示是可变式分区的一个示例。

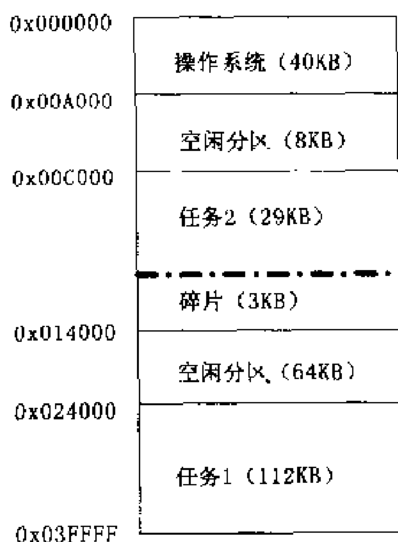


图 5-1　固定式分区

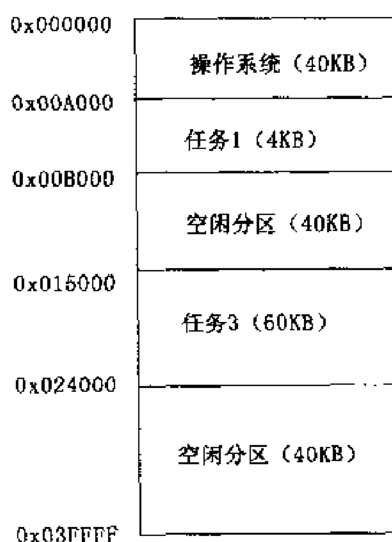


图 5-2　可变式分区

（1）可变式分区的内存的分配和回收。

在系统运行之初，操作系统空间（40KB）以后的全部内存被创建为一个大的空闲分区。每次申请内存，系统从该空闲区中划分出一块与程序大小相同的区域进行分配。其过程如下：

首先，在未分配分区的状态表中查找一个空白区，该区的大小必须大于或等于该次申请的内存大小。如果是前者，则将该分区分成两块，一块分配给申请的作业，另一份再次加入到空白分区状态表中。

反之，当一个任务运行完成之后，将释放该任务所占用的内存，即将该任务占用的内存重新加入到空白分区表中。分区回收时，应该检查回收区与内存中前后空闲区是否相邻，若相邻，则应进行合并，以便形成较大的空闲区。

根据分配内存时查找空白区顺序的不同，有着不同的内存分配算法。常用的分配算法有以下 3 种：

● 首次适应算法

这种算法把空闲分区按其在存储空间中地址递增的顺序链接在一起。用户申请内存空间时，从空闲区链表的头指针开始查找，选择第一个满足要求的空闲分区。如果该分区容量大于申请的大小，将其分成两块，一块分配给任务，另一块重新插入空闲区链表中。

该算法的优点是分配和回收算法都比较简单，查找速度快，因为这个算法总是从低地址开始查找，因此留在高地址部分的大空闲区被划分的机会很少，这样在大任务到来时很容易满足需要。

● 最优适应算法

该算法把空闲分区链表按分区大小由小到大进行组织。每次进行内存分配时，都是从空闲区表中找到一块满足进程需求的最小空闲区分配给它。这种做法减少了对大空闲区进行多次分割造成的空间浪费，有利于大作业的装入。

该算法会形成一些内部碎片，造成对内存空间的浪费。另外，每次分配时，都要对整个空闲区链表进行搜索，从而增大系统开销。

● 最坏适应分配算法

每次进行内存分配时，都从空闲区表中找到一个满足长度要求的最大空闲区进行分配，使得剩下来的空闲区不致太小。这种算法克服了由内部碎片引起的浪费，适合于中小作业的运行，但对大作业的运行是不利的。

与最优适应分配算法一样，该算法也需要对整个空闲区链表进行搜索，其效率会受到一定影响。

上述 3 种算法各有利弊，系统可以根据需要选用。

（2）可变式分区的碎片整理。

当系统运行一段时间后，随着一系列的内存的分配和回收，原来的一整块大空闲区间形成了若干已使用区和空闲区相间的布局。经过长时间的运行，内存中不能再分配利用的小碎片会越来越多。有时可能会出现这种情况，即当一个任务申请一定数量的内存时，虽然此时空闲区的总和大于该任务的内存要求，但是没有单个的空闲区域足以装下该作业。

解决这个问题的办法之一是采用紧凑技术，即把小碎片集中起来使之成为一个大的分区。实现的方法是移动用户分区中的程序，使它们集中于内存的一端，而使碎片集中于另一端，从而将空闲的碎片连成一个较大的分区，供需要的任务使用。

为了支持用户程序的移动，必须采用动态重定位技术。使用该技术需要得到硬件的支持，一般是在重定位机构中增加重定位寄存器。前面已经介绍过该技术，这里就不重复了。碎片整理的另一个缺点是整理工作需要占用处理器时间。

5.1.4 基于段、页的存储管理

早期基于分区的存储管理方案有一个严重的缺点：内存使用仍不充分，并且存在严重



的碎片问题。虽然采用碎片整理技术可以解决碎片问题，但会因为移动大量数据而浪费许多处理器时间。

引发该问题的根本原因是存储器使用的连续性，即系统对每个程序都分配一片连续的内存区。如果内存空间中没有能够满足要求的连续区域，即使可用内存空间的总容量大于进程需求量，系统也不能实施分配。

如果能够将一个任务的内存直接离散地分配到多个不相邻的分区中，就不必再进行碎片整理了。基于离散分配这一思想，产生了 3 种具体的分配方式：

- 分页存储管理。
- 分段存储管理。
- 段页式存储管理。

1. 分页存储管理

分页存储管理是一种离散分配存储管理方案，其基本出发点是打破存储分配的连续性。分页存储管理把系统中用到的地址空间分为两种。提供给用户程序使用的是基于逻辑地址的连续的地址空间，系统使用的是真实物理地址；由系统负责将连续的逻辑地址映射到物理上并不连续的实内存地址。

分页存储管理把内存空间分成大小相等、位置固定的若干个小分区，每个小分区为一个存储块，简称物理块或者页框，并依次编号为 0、1、2、3、…、n 块，物理块的大小由不同的系统决定，一般为 2 的 n 次幂，如 512B、1KB、2KB、4KB 等。

用户任务的逻辑地址空间同样分成与存储块大小相等的若干页，依次编号为 0、1、2、3、…、m 页。当任务提出存储分配请求时，系统首先根据存储块大小把任务分成若干页。每页都装入一个物理块中。此时，物理块并不要求连续，只要建立起程序的逻辑页和内存的存储块之间的对应关系，借助动态地址变换，原本连续的用户程序在分散的不连续存储块中就能够正常投入运行。

如果作业所需的页面一次全部装入到内存中，则称为纯分页存储管理。反之，如果作业所需页面根据作业运行时的实际要求分次装入，则称为请求式分页存储管理。

为了使不连续的、分散的用户程序能正常运行，操作系统必须处理逻辑地址和物理地址间的转换问题。通常系统在内存中为每个任务开辟一块内存区域，建立起任务的逻辑页与物理存储块之间的对应表格关系，该表称为页面映射表，简称页表。页表中的一项就对应任务的一个页面。页表项至少应该包含页号、块号两个内容。

利用页表结构，逻辑地址实际上被分为两个部分，页号和页内偏移量。其结构如图 5-3 所示。

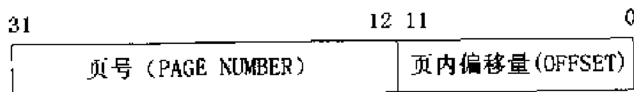


图 5-3 分页存储管理逻辑地址

该图说明了一个 32 位计算机的逻辑页地址结构。逻辑地址的高 20 位是页号，表明一

个任务可以有 1M 个页面，逻辑地址的低 12 位是页内偏移量，12 位地址可以寻址 4KB 的空间，即页面的大小是 4KB。

在任务执行过程中，由硬件地址分页结构自动将每条程序指令中的逻辑地址解释成两部分，页号 A 和页内地址 B。通过页号查找页表得到块地址 C，与页内偏移量 B 相加，形成物理地址，访问内存得到操作数据，其过程如图 5-4 所示。

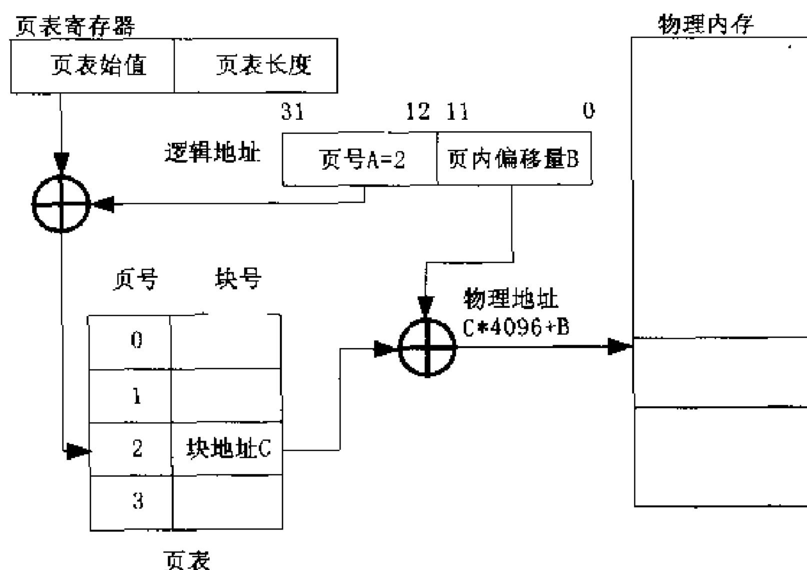


图 5-4　分页存储管理的地址转换（页面大小 4KB）

另外，为了提高查表的速度，人们在分页地址变换机构中加入一组高速缓冲存储器，用来存放当前作业最常用的页号和与之相应的物理块号。一般称这样的寄存器组为快表或联想存储器。因为联想存储器硬件成本较高，所以其数目不可能太多。

分页存储管理机制从根本上解决了连续存储分配带来的问题，消除了碎片，便于多道程序设计，从而提高了处理器和主存的利用率。但是分页管理仍然存在一些缺点：

- 动态地址转换将增加计算机成本和降低处理器的速度。
- 页表的创建和管理需要占用部分内存和处理器时间。
- 虽然消除了分区级的碎片，但是任务的最后一页一般都有页内碎片。例如，假定页面大小为 4KB，如果某一任务需要 9KB 内存，则必须为它分配 3 个物理存储块，最后一块中将有 3KB 的页内碎片。
- 仍然没有解决存储空间的“扩充”问题，即如何在内存中运行超过内存大小的程序的问题。

2. 分段存储管理

现代程序设计中，程序员一般都希望把信息按内容和逻辑关系分成多个部分，每个部分都有自己的名字，且可以根据名字来访问相应的信息块。分段存储管理机制就实现了这种模式。分段机制需要编译程序的支持，目前，大多数流行的编译器都支持分段，它们往往将程序划分为代码段（code segment）、数据段（data segment）、未初始化数据段（bss



segment）和其他一些附加的信息段等。

分段机制对于模块化程序和变化的数据结构的处理，以及不同作业之间对某些公共子程序或数据块的共享及保护等机制的实现，都提供了有力的支持。

（1）分段原理。

分段存储管理将内存空间动态地划分为若干个长度不相同的区域，每个区域都称为一个物理段。每个物理段在内存中都有一个起始地址，通常称为段首址。同时，将物理段中的所有单元从 0 开始依次编址，通常称为段内地址。

用户程序也按照其逻辑功能划分为不同的部分，通常称为逻辑段。例如主程序、子程序、数据等都可各成一段，每个逻辑段都对应于一个过程、一个程序模块或一个数据集合。逻辑段的长度就是其包含的信息的长度。因此，各个逻辑段的长度也是动态的。

当任务装入内存时，系统首先根据各逻辑段的大小，把物理内存分成相应的物理段，然后将逻辑段装入。此时，物理段的内存也并不要求连续，只要建立起程序的逻辑段和内存的物理段之间的对应关系，借助动态地址变换，原本连续的用户程序也能在分散的不连续存储块中，就能够正常投入运行。可见，分段管理也实现了内存的离散分配。

（2）分段存储管理的地址转换。

分段存储管理机制，使用段映射表来存放程序的逻辑段和内存的物理段之间的对应关系。段映射表又称为段表，一般常驻内存中。

类似于分页管理机制，分段系统中所使用的地址也被分成了两个部分，如图 5-5 所示。

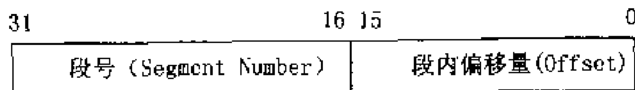


图 5-5　分段存储管理地址结构

该图示意了一个 32 位计算机的逻辑段地址结构。逻辑地址的高 16 位是段号，这表明一个任务可以有 64K 个逻辑段；逻辑地址的低 16 位是页内偏移量，16 位地址可以寻址 64KB 的空间，即段长度的上限是 64KB。

在任务执行过程中，由硬件地址分段结构自动将每条程序指令中的逻辑地址解释成两部分，段号 A 和段内偏移量 B。通过段号查段表得到段首址 C，与段内偏移量 B 相加，形成物理地址，访问内存，得到操作数据，其过程如图 5-6 所示。

不难看出，分页和分段存储管理有许多相似之处，所以很容易将两者搞混，但是这两者在概念上是完全不同的。分页的任务地址空间是一维的线性地址空间，而分段的任务地址空间是二维的。页的大小固定（例如 1KB 或 4KB），段的长度是动态的。分页对用户是透明的，而分段是用户可见的，页是信息的物理单位，使用分页机制是为了实现离散分配，以减少内存的碎片。段是信息的逻辑单位，而使用分段是为了方便用户的编程。

（3）分段的好处。

另外，为了提高查表的速度，人们在分段地址变换机构中加入一组高速缓冲存储器，用来存放当前作业最常用的段号和与之相应的段基址、段长等信息。它通常是作为段寄存器的影子寄存器出现，其缓冲过程由处理器自动完成，对用户透明。一般情况下，一个段

寄存器配备一个高速缓冲存储器。

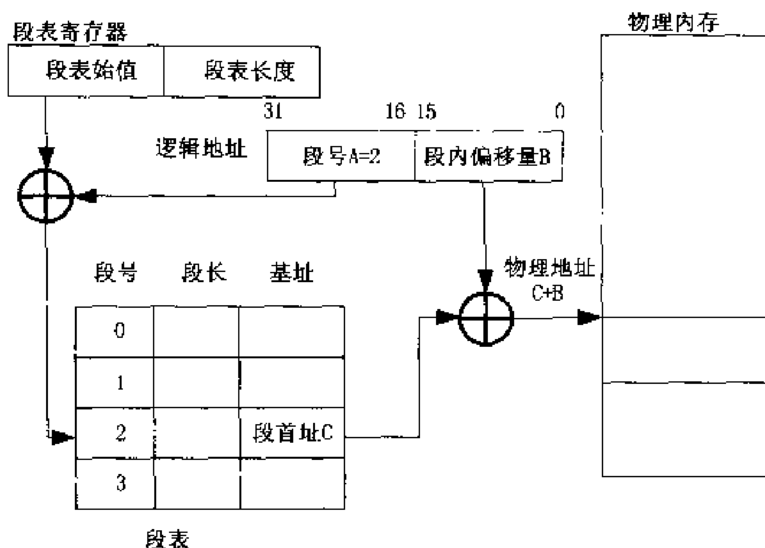


图 5-6 分段存储管理的地址转换

3. 段页式存储管理

分页存储管理方式和分段存储管理方式都有各自的优缺点，分页系统能有效地提高内存的利用率，而分段系统则能方便用户程序的逻辑组织。如果综合使用这两种方法，则可以形成一种新的存储管理方法，该方法兼有分页存储管理方式和分段存储管理方式的优点，通常把这种存储管理方式叫做段页式存储管理。这种方式既可以很好地解决内存的碎片问题，也可以为用户编程提供支持。

（1）基本原理。

在段页式存储管理中，首先将用户的任务空间分成若干个段，再把每个段划分成若干个页。为了装入用户任务，内存空间也被划分为相同大小的页面，多个页按照段的长度组织在一起，可以装入用户对应段的信息。

本质上说，段页式存储管理是用页式方法来分配和管理内存空间，用段式方法对用户程序进行逻辑划分。与前面单纯分段机制所不同的是，段页式存储管理中的段，物理上并不一定是连续的，而且可以说绝大多数情况下，都不是完全连续的。

当任务装入内存时，对于任务的每个逻辑段，系统都会读取该段的大小，计算出该逻辑段需要多少个物理页，然后为该段建立一个页表，记录该段的所有页面，然后将该段的信息装入到这些页面中。此时，不仅段之间不需要连续，段内的页面也不要求连续，只要建立起程序的逻辑段与段内的页面，以及段内页面和物理页之间的对应关系，借助动态地址变换，原本连续的用户程序也能在分散的不连续存储块中，就能够正常投入运行。可见，段页式管理也实现了内存的离散分配，并且其灵活性更强。

（2）段页式存储管理的地址转换。

为了处理逻辑地址和物理地址间的转换，段页式存储管理机制需要同时使用段映射表和页映射表。系统必须为每个程序建立一张段表，由于一个段又被划分成了若干页，系统



又必须为每个段建立一张页表。段表中记录了该段对应页表的起始地址和长度；而页表则给出该段的各个逻辑页面与内存块号之间的对应关系。

使用这种方法，段页式系统中所使用的地址也被分成了 3 个部分：段号、页号和页内偏移量，如图 5-7 所示。

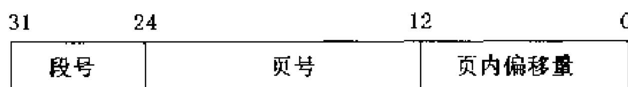


图 5-7　段页式存储管理地址结构

该图说明了一个 32 位计算机的逻辑段地址结构。逻辑地址的高 8 位是段号，这表明一个任务可以有 256 个逻辑段；逻辑地址的中间 12 位是页号，表明一个逻辑段可以有 4K 个页；逻辑地址的低 12 位是页内偏移量，表明页的大小也是 4KB。

在任务执行过程中，由段页式硬件地址转换结构自动将每条程序指令中的逻辑地址解释成 3 部分：段号、页号和页内偏移量 B。通过段号查找段表得到页表首址，再次通过页号查页表得到块首址 C。最后将块首址与段内偏移量 B 相加，形成物理地址，访问内存，得到操作数据，其过程如图 5-8 所示。

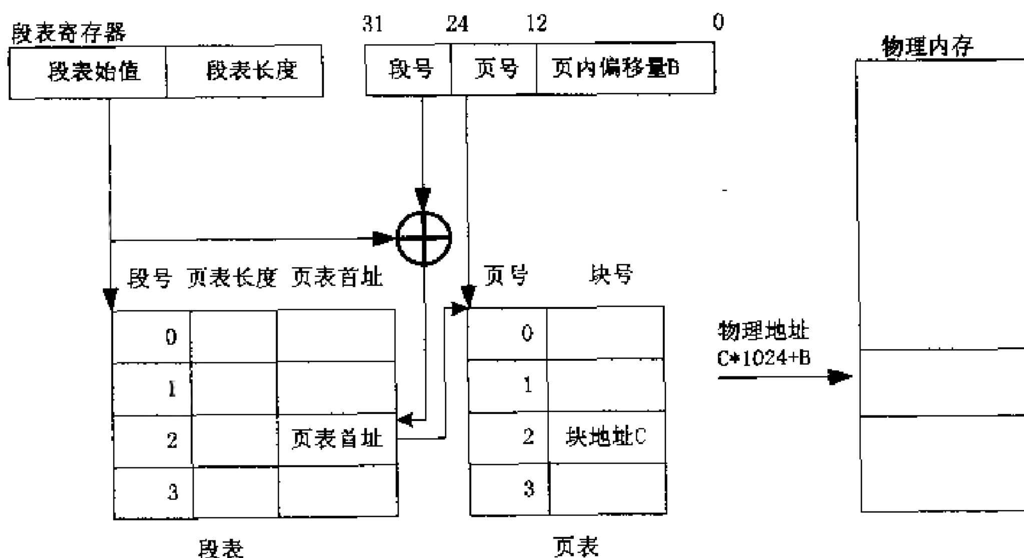


图 5-8　段页式存储管理的地址转换

段页式存储管理方案吸取了分段存储管理和分页存储管理的优点，能有效地利用主存，为组织多道程序运行提供了方便。其主要缺点是增加了硬件成本、系统的复杂性和管理上的开销；存在页内碎片，各种管理数据结构（如段表、页表）需要占用主存空间。

5.1.5　虚拟存储器管理

本章前几节介绍的各种存储管理方案有一个共同的问题，即当某个进程运行时，其整

个程序必须都已经装载到内存中。此类方法存在如下缺点：若一个进程的程序比内存可用空间还大，则该程序无法运行；由于程序运行的局部特性，一个进程在运行的任一阶段只需使用所占存储空间的一部分，因此，未用到的内存区域就被浪费了。

现在，问题就变成了是否必须将整个程序都装入内存才能开始执行？如果不是，那么如何在只装入部分程序的情况下，保证程序的正确执行。这就是虚拟存储器管理所要解决的问题。

1. 程序局部性原理与请求调页

要想解决存储空间的“扩充”问题，必须利用程序局部性原理。理论和模拟试验都表明，在冯偌依曼结构的计算机中，几乎所有的程序在执行过程中都表现出一定的局部性。局部性可以分为时间局部性和空间局部性。

- 时间局部性：一条指令可能被反复执行；某个数据结构可能被反复访问，这样的指令和数据结构就体现了时间局部性。
- 空间局部性：若某一存储单元被使用，那么与该存储单元相邻的单元可能也会立即被使用。程序代码的顺序执行，对线性数据结构的访问或处理，以及程序中往往把常用变量存放在一起等都反映出空间局部性。

程序局部性原理，说明了没有必要一次性把整个程序全部装入内存后再开始运行，在程序执行过程中其某些部分也没有必要从开始到结束一直都驻留在内存中。

2. 虚拟存储器基本原理

基于局部性原理，一个作业在运行之前，没有必要全部装入内存，而仅将那些当前要运行的那部分页面和段先装入内存便可以开始运行。在程序的运行中，发现所要访问的段不在内存中时，再由操作系统将其调入内存，程序便可继续执行下去。

这样，在运行过程中，程序只需要部分装入内存，其余部分可以存放在外存，等到确实需要时，再调用到内存中。对用户来说，看到的是一个很大容量的内存，可以同时运行多道的任务。利用局部性原理，把内存与外存有机地结合起来使用，从而为用户提供一个容量很大的、速度足够快的“内存”，这就是虚拟存储器，简称虚存。

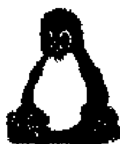
影响虚拟存储器容量的因素包括两个方面，首先是计算机硬件所能提供的最大地址空间，这个地址空间一般受地址线数目的限制；其次，外存的大小也限制了所能交换的地址空间。多数情况下，外存大小往往是限制虚拟存储器容量的主要因素。

3. 虚拟存储器的实现

虚拟存储器的实现，都是基于离散分配方式实现的。虚拟存储技术主要分为虚拟页式存储管理和虚拟段式存储管理两种。

- 虚拟页式存储管理

虚拟页式存储管理是在分页存储管理的基础上，增加了请求调页功能、页面置换功能所形成的页式虚拟存储系统。它以页面为基本单位，允许在只装入部分页（有一个最小值，该最小值由一条指令最多所能访问的页面数决定）的情况下，开始程序的运行。在程序运



行中，动态地调入和换出页面，以保证程序的正确执行。

- 虚拟段式存储管理

虚拟段式存储管理在分段存储管理的基础上，增加了请求调段功能、分段置换功能所形成的段式虚拟存储系统。它以段为基本单位，允许在只装入部分段的情况下，开始程序的运行。在程序运行中，动态地调入和换出段，保证程序的正确执行。

虚拟分段和虚拟分页系统的实现都需要一些额外的支持，其主要内容如下：

- 扩充管理数据结构。为了支持请求调页（或段）功能，需要在页表（或段表）中添加若干数据项。
- 缺页/段中断机制。当用户程序访问当前不在内存中的页面时，会引起缺页（或段）中断，以请求操作系统将所缺的页面（或段）调入内存。
- 地址变换机构。扩充分页（或分段）的地址变换机构，以支持动态调页功能。

通过以上分析，可以看到，两种管理方式的原理大同小异，下面主要以虚拟页式存储管理为例来进行更详细的介绍。

（1）页面的调入与换出。

由于内存大小的限制，在程序运行过程中，虚拟存储器管理会出现两种需要特殊处理的情况。一种是所要访问的页当前不在内存中，需要将其从外存中调入；另一种是在向内存中调入一个页面时，发现内存空间不足，这时需要将内存中一些页面交换到外存中，以便腾出空间来调入新的页面。这两种操作分别叫做页面的换入与换出。

页面的换入通过缺页中断机制实现。当程序指令访问的操作数不在已经装入的页面的范围中时，CPU 会触发一个硬件中断，通知操作系统，当前发生了一个缺页事件。操作系统对应的中断处理程序则负责从外存中调入该操作数所在的页面。缺页中断实际上是发生在指令执行的过程中，并且由于一个指令可能有多个操作数，因此在一条指令的执行过程中，可能发生多次缺页中断。

页面的换出既可能是操作系统的定期行为，也可能是由用户的页面调入请求引起的。它选择内存中的某些页，将其标记为未装入，然后释放其空间，以便装入新的页面。页面换出时最大的问题在于如何选择被换出的页面。把进行这种选择的算法叫做页面置换算法。

（2）页面置换算法。

目前常见的页面置换算法包括最优置换算法、LRU、FIFO、NRU 等。

- 最优置换算法

该算法选择将来最长时间不使用的页面换出，可以获得最好的性能。但是由于无法动态预测一个程序访问内存页面的情况，所以该算法是不可实现的。主要用于估计其他算法的性能。

- LRU 算法

LRU 算法又称为最近最久未使用算法。它选择过去时间内最久未使用的页面换出。该算法试图用“最近的过去”代表“最近的将来”。理论和试验表明，LRU 算法较好地模拟了最优置换算法，有较好的性能。目前，大多数的请求调页系统都采用了 LRU 算法。

- FIFO 算法

FIFO 算法又称为先进先出置换算法。它选择最早进入内存的页面换出。该算法的优点

是实现简单，但是其性能不好。因为最早进入内存的页面并不一定是使用最少的。

（3）虚拟页式存储的地址转换。

虚拟页式存储管理与纯分页存储管理在内存块的分配、回收、存储保护等方面都十分相似，主要的不同之处在于地址重定位问题。在虚拟页式存储管理的地址重定位时，可能会出现所需页面不在主存的情况，此时系统必须解决以下两个问题：

- 当程序要访问的某页不在内存时，如何发现这种缺页情况?发现后应如何处理?
- 当需要把外存中的某个页面调入内存时，此时内存中没有空闲块怎么办?

图 5-9 显示了虚拟页式存储的地址转换过程。在地址变换时，首先计算出指令或操作数的地址所在的页面，然后查询页表，判断该页是否在内存中。如果在，则修改页表中的访问位和修改位，然后执行对应操作。如果不在，则产生一个缺页中断。缺页中断的处理过程如下：

首先保存当前任务的上下文，然后在外存中查找所缺的页面。接着判断内存中是否还有空闲空间可以装入该页面，如果有，则直接调入该页面，然后修改页表，中断返回，执行该指令。如果没有，则使用某种页面置换算法，在内存中选出部分页面换出，然后再调入所缺的页面。

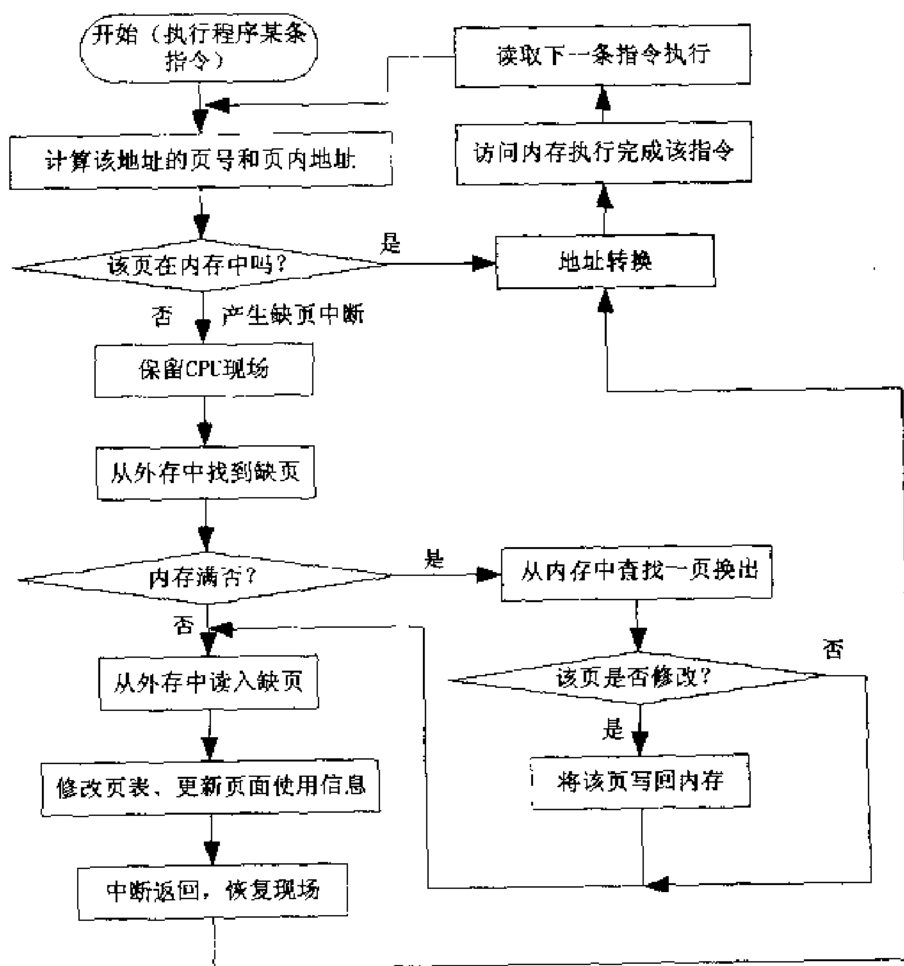
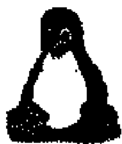


图 5-9　虚拟页式存储的地址转换



（4）虚拟页式存储的优缺点。

虚拟页式存储管理，继承了页式存储管理的优点，特别是消除了内存碎片问题。另外，它还提供了大容量的虚拟存储器，使得任务的地址空间不再受实际物理内存容量的限制；它更有效地利用了主存，任务的地址空间不必全部同时都装入主存，只装入其必要部分就可运行；它更加有利于多道程序的运行，从而提高了系统吞吐量。

5.2 Linux 存储器管理

存储器管理是 Linux 操作系统的重要组成部分，为 Linux 系统其他模块的实现提供了有力的支持。Linux 实现了基于虚拟页式存储管理的虚拟存储，在 i386 结构的机器上，每个用户任务的虚拟地址空间都可达到 4GB。

Linux 是一个通用的跨平台操作系统，因此 Linux 的实现必须考虑到在不同硬件平台间的可移植性。很多微处理器都有专门的存储器管理单元（MMU），但是其实现没有（也不可能有）统一的标准。比如，Alpha AXP 处理器上页面大小为 8KB，而 Intel x86 处理器上页面大小为 4KB；Intel x86 使用两级页表，Motorola M68K 系列使用三级页表。为此，Linux 的存储管理使用了三层页表来处理逻辑地址到物理地址的转换，分别是 PGD（页表目录）、PMD（中间页表目录）和 PT（页表）。

存储管理是非常复杂的工作。Linux 将存储管理分为物理内存管理、内核内存管理、虚拟内存管理、内核虚拟内存管理和用户级内存管理。

- 物理内存管理

物理内存管理以页为单位，记录、分配和回收物理内存，物理内存管理使用 Buddy（伙伴）算法。

- 内核内存管理

内核内存管理主要负责为各种内核数据结构分配空间，其大小一般较小。如果使用以页为单位的物理内存管理则浪费较大，为此 Linux 专门提供了使用 Slab 算法的内核内存管理。

- 虚拟内存管理

在物理内存管理的基础上，使用请求调页机制和交换机制，为系统中的每个进程都提供高达 4GB（i386 平台）的虚拟内存空间。

- 内核虚拟内存管理

Linux 将每个进程的 4GB 虚拟内存分为用户区（0~3GB）和内核区（3~4GB），内核虚拟内存管理负责内核区虚拟内存的管理。

- 用户级内存管理

用户任务空间，即进程的内存空间管理，一般由 C 库实现，目的是为用户的应用编程提供支持。

5.2.1 物理内存管理

1. 空闲物理内存单元的管理

Linux 物理内存管理使用 Buddy 算法实现。其物理页面的信息由 mem_map_t 结构描述，系统中的所有物理页面由一个 mem_map_t 类型的数组 mem_map 来表示。该数组的每一项都代表物理内存的一个页面，该数组的大小由实际的物理内存大小决定。mem_map_t 结构的定义如下（摘自 linux\linux-2.4.x\include\linux\mm.h）：

```
typedef struct page {
        struct page *next;
        struct page *prev;
        struct inode *inode;
        unsigned long offset;
        struct page *next_hash;
        atomic_t count;
        unsigned long flags;
        struct wait_queue *wait;
        struct page **pprev_hash;
        struct buffer_head * buffers;
} mem_map_t;
```

其重要字段如表 5-1 所示。

表 5-1 mem_map_t 结构重要字段

next,prev	指向 mem_map_t 结构的指针，维护了空闲物理页的双向链表
inode、offset	如果该物理页用于存放来自文件的数据时，本页面用于存放数据或代码所属的文件的 inode 和偏移量
next_hash、pprev_hash	指向 mem_map_t 结构的指针，维护了在页面 cache 的散列表中的一个双向链表
count	该物理页面的引用计数
flags	标志位，表示页面的当前状态
wait	因等待操作该物理页面而阻塞的进程的队列

为了记录系统中的当前空闲物理内存单元，Linux 内核定义了 free_area 数组。该数组每一项都是一个 free_area_struct 结构，描述了一组由相同大小的空闲物理页块构成的双向链表。在 free_area[0]上是大小为 1 页的空闲页块链，free_area[1]上是大小为 2 页的空闲页块链，free_area[2]上是大小为 4 页的空闲页块链，依此类推，以 2 的指数递增。free_area_struct 结构定义如下：




```
struct free_area_struct {
        struct page *next;
        struct page *prev;
        unsigned int * map;
        unsigned long count;
};
```

其中指针 next、prev 用于将空闲物理页块结构 mem_map_t 链接成一个双向链表，而 map 则是一个位图，用于记录页块及其伙伴是否在本队列中。该位图中的位标识了该队列中某一块的两个伙伴的情况。该位为 0 表示两个伙伴都不在链表中，为 1 表示有且只有一个伙伴在链表中。如果某块的两个伙伴都在该链表中，则它们应该合并成更大的块，加到 free_area 数组中下标值更大的数组项所指的链表中。图 5-10 说明了 Linux 空闲物理块的管理方法。

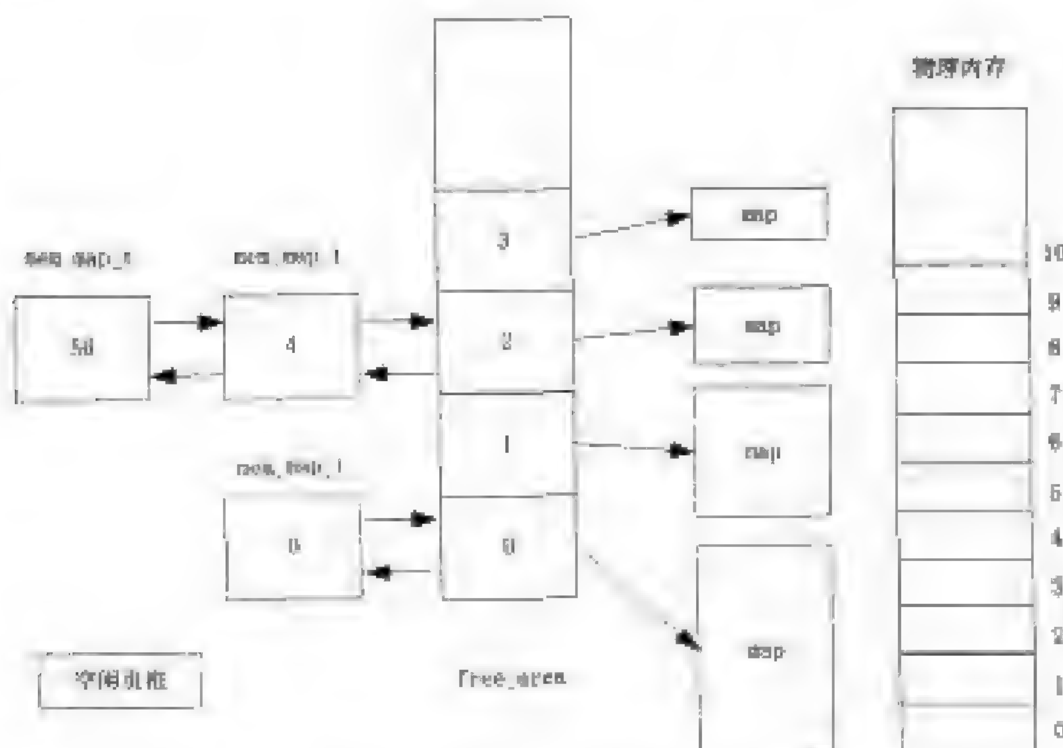


图 5-10　Linux 空闲物理页块的管理

另外，Linux 使用全局变量 nr_free_pages 来跟踪系统中的空闲物理内存页面的总数，该变量的值等于 free_area 数组中空闲页的总数。使用该变量，Linux 可以知道系统物理内存的使用状况，当空闲的物理内存页面数量低于某个指标时，Linux 将唤醒内核交换守护进程 kswapd，让其尽可能回收一些物理内存。Linux 试图将系统中的空闲物理页面数维持在一个特定的数量上，该数量由 freepages 结构指定，其定义如下：

```
typedef struct freepages_v1{
        unsigned int        min;
        unsigned int        low;
        unsigned int        high;
}freepages_v1;
```

其中 min 字段就是系统应该维护的最少空闲物理页面数。

2. 物理页的分配

前面已经提到，Linux 使用 Buddy 算法来分配和释放物理内存。因为不能像逻辑地址一样进行映射，所有物理内存的分配和释放必须针对连续的物理内存。因此，Buddy 算法将内存划分成 2^i 页大小的连续的页块，每一组有相同的 i 值，并且将同一组的空闲页块链接成一个双向链表，然后将所有链表的头节点组成一个数组，这就是上面的 free_area 数组。free_area 的数组已经在上面介绍过了，其结构如图 5-10 所示。

为了避免内存碎片的产生，系统每次分配的空间都是 2 的整数次幂个页面，如果申请的空间大小不是 2 的整数次幂，则会将一部分多余的空间分配给进程，这也是 Buddy 算法的一个缺点。

在分配空间时，首先要在页块大小相应的空闲页块链中找到一块空间来返回给调用者，如果在这个链里没有找到合适的空间分配，则到 free_area 数组中的下一个元素的空闲链中去查找，直至找到能够分配的空间，然后，将得到的空闲块进行分割，直至得到的块大小与申请的块大小相匹配，再将那些分割出来的空闲块插入到相应的空闲链中去。这一过程是由函数_get_free_pages()（在 linux\linux-2.4.x\mm\page_alloc.c 文件中）来实现的。_get_free_pages()函数又间接调用了_alloc_pages()函数来实现具体的分配操作。感兴趣的读者可以自行参考其源代码。

3. 物理页的释放

内存管理系统，在分配空间时将大页块都划分成了小页块，这使得系统中的页块都越变越小，这对于分配大块内存是不利的。为了避免产生大量的内存碎片，内存管理系统在物理页释放时，应该尽可能地将小页块合并成大页块。

首先，内存管理系统会检查当前释放的内存块的伙伴是否在空闲链表中。如果在，就将它们合并为一个两倍大小的空闲块，然后再继续查找两倍大小块的伙伴，重复上述过程直至不能合并为止。函数 free_pages()用于释放页块（在 linux\linux-2.4.x\mm\page_alloc.c 文件中），同时 free_pages ()函数又间接调用了_free_pages_ok ()函数来实现具体的释放回收操作。感兴趣的读者可以自行参考其源代码。

4. 物理内存管理的评价

为了内存对齐，Buddy 算法往往会分配一些多余的内存空间给用户，这对提高内存利用率是不利的；但是 Buddy 算法避免了内存碎片的产生，避免了内存紧凑的开销，效率很高。



5.2.2 虚拟内存管理

1. 进程的虚拟地址空间

Linux 在分页系统的基础上实现了虚拟内存管理。在 i386 平台上，每个进程各自拥有完全独立的 4GB 虚拟地址空间。其中 0~3GB 是用户空间，3~4GB 是内核空间。任务各自私有的代码和数据都存储在用户空间，Linux 内核自身的代码和数据则存储在内核空间，由所有任务共享，如图 5-11 所示。注意，该图中右边的地址空间只是一种示意，实际上由于 Linux 采用分页机制，右边的地址空间不可能都是连续的。

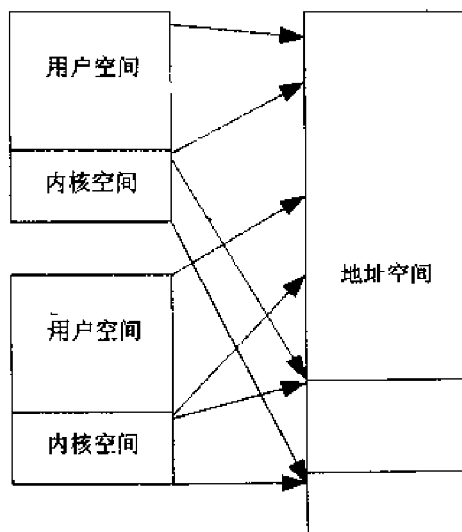


图 5-11　用户空间和内核空间映射示意图

2. 页表的管理

一个页表入口标识一个物理页，它包含了物理页的大量信息，如该页是否有效、该页的读写权限等。最重要的是页表入口给出了物理页的页框号（PFN），根据这个物理页框号就可以找到这个物理页的实际起始物理地址。

前面已经提到，虚拟存储管理系统中，所有进程都使用虚拟地址。但是，为了能够正确地读取代码和操作数据，CPU 必须将虚拟地址转换为物理地址。在转换的过程中，MMU（存储器管理单元）需要读取一组由操作系统维护的表格，包括页表和页目录等（和具体的 CPU 有关），它们存储了虚拟地址到物理地址的映射关系。

为了操作系统的可移植性，Linux 使用三级页表来存储这种映射关系。一级页表只占用一个页，其中存放了二级页表的入口的指针，记为 PGD；二级页表中存放了三级页表的入口的指针，记为 pmd；在三级页表中每个项都是一个页表入口（pte）。但是，根据不同的处理器平台，所有的三级页表并不一定都被使用。实际上，在 Linux 的 X86 版本中，只使用了两级页表，即第一级和第三级，第二级页表实际上并没有使用。在 Intel 系列 CPU 中，

一个物理页面大小是4KB，而每个页表项大小是4字节。因此，每个物理页面可以包含1024个页表项，则在X86平台上，每个进程的地址空间为1024×1024×4KB=4GB大小。

在X86平台上，在mm_struct结构中（用于描述一个进程的虚拟内存）存储了指向一级页表的指针pgd，而进程的二级页表和三级页表都是在任务运行过程中动态创建的。

3. 虚拟存储空间的管理

在Linux系统中，主要使用了3个层次的数据结构vm_area_struct、mm_struct和page来表示进程的虚拟地址空间。

最底层的page结构描述了一个物理页帧及其页内信息的相关属性和链接指针，包括标志位、引用计数等。该结构已经在前面介绍过，读者可以参考5.2.1小节的内容。

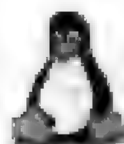
vm_area_struct结构是中间层次，它描述了一个虚拟内存区域（即一段连续的虚拟地址空间）的属性。其中，包括虚拟内存区域的开始、结束地址、访问权限、页目录、映射文件和链接指针。其定义如下（摘自linux\linux-2.4.x\include\linux\mm.h）：

```
struct vm_area_struct {
        struct mm_struct * vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;
        struct vm_area_struct *vm_next;
        pgprot_t vm_page_prot;
        unsigned long vm_flags;
        short vm_avl_height;
        struct vm_area_struct *vm_avl_left;
        struct vm_area_struct *vm_avl_right;
        struct vm_area_struct *vm_next_share;
        struct vm_area_struct **vm_pprev_share;
        struct vm_operations_struct * vm_ops;
        unsigned long vm_pgoff;
        struct file * vm_file;
        unsigned long vm_raend;
        void * vm_private_data;
};
```

其重要字段如表5-2所示。

表5-2 vm_area_struct结构数据说明

Vm_start,vm_end	虚拟内存区域在进程虚拟地址空间中的开始和结束位置
Vm_next	指向vm_area_struct结构的指针，用于将同一进程的所有虚拟内存区域链接成一个单链表
Vm_file	指向file结构的指针，记录了该虚拟内存区域映射的文件



（续）

Vm_pgoff	表示该虚拟内存区域映射的内容在映射文件中的偏移位置，以 PAGE_SIZE 为单位
Vm_next_share、vm_pprev_share	指向 vm_area_struct 结构的指针，用于将多个 vma 结构链接成双向链表。当该 vma 用于映射文件时，这两个指针将同一文件的所有 vma 结构链接起来；当该 vma 用于映射共享内存时，它们将所有映射该内存区域的 vma 链接起来
Vm_avl_left、vm_avl_right	指向 vm_area_struct 结构的指针，用于将 vma 链接成一棵 AVL 树
Vm_flags	描述该区域的属性
Vm_page_prot	描述该区域的页的默认保护权限（读、写、执行）
Vm_ops	该区域的处理操作集（包括页错误处理、页换入、页换出等）

在进程的虚拟地址空间内查找一个特定的 vm_area_struct 结构，是使用频率较高的操作。为了提高查找的效率，Linux 同时将一个进程的 vm_area_struct 结构链接成了单链表和 AVL 树。用单链表来管理 vm_area_struct 结构的优点是在节点数比较少的时候可以很快地检索到需要的节点，而且插入和删除节点都很方便；其缺点就是随着进程虚拟地址空间不断扩大，vm_area_struct 结构的节点数将不断增加，其检索效率会大大下降。用 AVL 树来管理 vm_area_struct 结构的优点是检索效率高，尤其是在 vm_area_struct 结构节点数很多的时候；缺点是建立 AVL 树需要额外代价，插入节点和删除节点都比单链表要麻烦。因此，Linux 将二者的优点结合了起来，在 vm_area_struct 结构节点数比较少的时候，就只采用单链表来管理 vm_area_struct 结构，而这时的 vm_avl_left、vm_avl_right 指针为空；随着进程的不断运行和 vm_area_struct 结构节点数的增加，当 vm_area_struct 结构节点数超过 AVL_MIN_MAP_COUNT（Linux 中定义为 32）时，就会为这个进程建立 vm_area_struct 结构节点的 AVL 树，这时的 vm_avl_left、vm_avl_right 指针不再为空。这样，Linux 将单链表的优点和 AVL 树的优点结合起来，既保证了拥有较高的查找效率又尽量减少了不必要的额外代价。

mm_struct 是描述进程虚拟地址空间的最高层的数据结构，一个 mm_struct 就代表一个独立进程的虚拟内存空间。该结构中记录了实现任务管理的进程模型所需要的内存管理相关的全部信息，如：

- 进程的页目录的位置。
- 进程的代码、数据、堆栈、堆、环境变量、入口参数等在虚拟地址空间中的存储位置。
- 进程占用的物理页帧数目、进程的 LDT（局部描述符表）、引用计数。
- 进程的虚拟地址空间的虚拟内存区域链表的链接信息和一些统计信息。

该结构的定义如下（摘自 linux\linux-2.4.x\include\linux\sched.h）：

```
struct mm_struct {
        struct vm_area_struct * mmap;
```

```
        rb_root_t mm_rb;
        struct vm_area_struct * mmap_cache;
        pgd_t * pgd;
        atomic_t mm_users;
        atomic_t mm_count;
        int map_count;
        struct rw_semaphore mmap_sem;
        spinlock_t page_table_lock;
        struct list_head mmlist;
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;
        unsigned long rss, total_vm, locked_vm;
        unsigned long def_flags;
        unsigned long CPU_vm_mask;
        unsigned long swap_address;
        unsigned dumpable:1;
        mm_context_t context;
};
```

其重要字段如表 5-3 所示。

表 5-3　mm_struct 数据结构说明

Mmap	指向 vm_area_struct 结构的指针，表示该进程的 vma 链的头指针，该链上所有的 vm_area_struct 结构按地址从小到大排列
mmap_cache	指向 vm_area_struct 结构的指针，指向上一次查找操作找到 vm_area_struct 结构，下一次查找首先检查该 vm_area_struct 结构是否满足条件
Pgd	指向进程页目录的指针
mm_users	使用该用户空间的用户数
mm_count	该结构的应用计数，当 mm_count 大于 0 时，该结构不能销毁
map_count	该进程的 vma 结构的个数
start_code, end_code	该进程代码在虚拟空间中的开始和结束位置
start_data, end_data	该进程数据在虚拟空间中的开始和结束位置
start_brk, brk	该进程堆在虚拟空间中的开始和结束位置
start_stack	该进程堆栈的栈顶在虚拟空间中的位置（栈底在 3GB 处）
arg_start, arg_end	该进程参数在虚拟空间中的开始和结束位置
env_start, env_end	该进程环境变量在虚拟空间中的开始和结束位置



（续）

Rss	该进程当前真实占用的物理页面数
Total_vm	该进程占用的总虚拟内存页数
locked_vm	该进程锁定的虚拟内存页数

从该结构可以看出，一个进程的虚拟地址空间被划分为多个虚拟内存区域，这种内存区域使用 vm_area_struct 结构来加以描述，一个虚拟内存区域描述一段连续的虚拟内存，它们有相同的属性和操作权限。一个进程的所有虚拟内存区域合在一起，就表示了该进程中所有有效的地址空间。

将进程的虚拟地址分为不同的区域，有几方面的好处。其一，实现了虚拟地址空间的离散分配，以方便用户将具有不同逻辑功能的信息装入不同的虚拟内存区域。例如，有的虚拟内存区域来自可执行文件的映像；有的来自共享库；有的来自动态分配的内存存储区。其二，利用 vm_area_struct 结构中的 vm_ops 操作集，Linux 还可以对不同的虚拟内存区域进行不同的处理，这实质上就是一种面向对象的思想，即不同的对象可以有不同的操作。其三，使用虚拟内存区域可以方便地表示出进程的有效地址空间，为检查非法地址访问提供了支持。

有了上述几个数据结构，Linux 系统可以方便地实现虚拟内存的管理。一个进程的虚拟地址空间在进程创建时即已创建，在进程销毁时释放。在进程的生命周期中，虚拟地址空间会不断变化。

4. 虚拟地址空间的创建

在第 3 章中，曾经介绍在 UNIX 类操作系统中，进程的创建和新任务的执行被分成两个单独的操作。当准备运行某个程序时，必须先调用 fork 系统调用，创建一个新进程，然后使用 exec 系统调用，将新的可执行映像装入到进程的虚拟地址空间。如果程序使用了某些共享库，则所有用到的库都必须同时装入。

这样进程虚拟地址空间的创建实际上可以分为两个步骤。首先，复制父进程的地址空间；然后，根据可执行映像的要求，创建新的内存地址空间。下面分别介绍进程虚拟地址空间的创建步骤。

（1）虚拟地址空间的复制。

复制进程的虚拟地址空间是 UNIX 操作系统任务机制的要求。但是，完全复制父进程的地址空间，需要将父进程的内存管理结构 mm_struct、vm_area_struct、页目录表、页表以及对应物理页帧的内容都复制到子进程中。显然，这样做的开销太大（特别是复制物理页帧的内容需要消耗大量的处理器时间和大量的物理内存），而且这有可能是不必要的（因为刚复制的地址空间有可能尚未使用就废弃）。

针对这个问题，目前有两种解决办法。BSD 系统提供了 vfork 系统调用，该调用将不复制地址空间，在子进程调用 exec()函数之前，父子进程将共享一个父进程的地址空间。并且父进程在调用 vfork()函数后就进入睡眠状态，直到子进程调用 exec()函数或是退出。

而子进程调用 exec()函数将重建新的虚拟地址空间。这样使用 vfork()函数就避免了不必要的虚地址空间复制，但是该技术需要程序员明确使用不同的系统调用。

SystemV 提供了另一种解决方法，即所谓的 Copy On Write（COW，写时复制）技术。执行 fork 系统调用后，子进程复制父进程的页目录和页表，但是不复制页面的内容，而将子进程的页表项指向父进程的物理页面，并且父子进程的物理页面都标记为只读。此后，当父进程或子进程修改某个页面时，将产生一个页面异常。而相应的异常处理程序在识别出这种情况后，会将产生异常的页面复制一份，并修改对应的页表项，使其指向新的物理页面，并去掉只读标志。使用 COW 技术，实质上是将页面的复制尽可能地延后，以避免不必要的复制操作。

而 Linux 综合使用了上述两种技术。对于 fork 调用，Linux 使用了 COW 技术，同时 Linux 也提供了 vfork 系统调用，使用 vfork 系统调用创建的进程完全和父进程共享同一个地址空间，包括页目录和页表。实际上，在 Linux 中，可以使用 vfork 系统调用实现线程管理。

下面讨论 fork 系统调用的虚拟地址空间复制。

fork 系统调用内部调用 copy_mm()函数实现虚拟地址空间的复制。该函数定义如下（摘自 linux\linux-2.4.x\kernel\fork.c）：

static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)

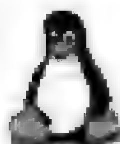
其中，clone_flags 参数表明复制进程虚拟地址空间的方法，tsk 参数是指向子进程的 task_struct 结构的指针。该函数首先初始化 tsk 中与内存管理相关的域（将 min_flt、maj_flt、cmin_flt、cmaj_flt、nswap、cnswap 数据项清零，mm、active_mm 数据项指向 NULL）；然后，为子进程分配一个 mm_struct 结构。如果分配成功，下面调用 mm_init()函数初始化该 mm_struct 结构，为其分配页目录表空间；最后，调用 dup_mmap()函数，复制进程的所有虚拟内存区域。下面是该函数的实现代码（摘自 linux\linux-2.4.x\kernel\fork.c）：

```
static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)
{
    struct mm_struct * mm, *oldmm;
    int retval;

    tsk->min_flt = tsk->maj_flt = 0;
    tsk->cmin_flt = tsk->cmaj_flt = 0;
    tsk->nswap = tsk->cnswap = 0;

    tsk->mm = NULL;
    tsk->active_mm = NULL;
    /*初始化 tsk 中（子进程）与内存管理相关的一些域*/

    /*
     * Are we cloning a kernel thread?
```




```
	 *
	 * We need to steal a active VM for that..
	 */
	oldmm = current->mm; /*使用 oldmm 指针保存当前进程（父进程）的 mm_struct 结构的位置*/
	if (!oldmm)
		return 0;

	if (clone_flags & CLONE_VM) {/*检查 clone_flags 是否设置了 CLONE_VM 标志，如果是，则说明该次内存复制操作由 vfork()函数引发，按照 vfork()函数调用规则*/
		atomic_inc(&oldmm->mm_users);
		mm = oldmm;
		goto good_mm;
/*直接将父进程的 mm_struct 结构（oldmm）返回给子进程，即父子进程共享相同的地址空间*/
	}

	retval = -ENOMEM;
	mm = allocate_mm();  /*如果没有设置 CLONE_VM 标志，表明该次操作由 fork 引发，需要为子进程创建新的地址空间，于是调用 allocate_mm()函数，为子进程创建 mm_struct 数据结构（mm）*/
	if (!mm)
		goto fail_nomem;  /*检查该次分配是否成功，如果不成功，则返回出错信息*/

	/* Copy the current MM stuff.. */
	memcpy(mm, oldmm, sizeof(*mm)); /*使用内存复制函数，直接复制 oldmm 中的信息到 mm，这样子进程就可以直接继承父进程的全部属性。接下来的工作，就是对父子进程 mm_struct 结构中不同的域进行处理*/
	if (!mm_init(mm)) /*调用 mm_init()函数，对子进程 mm_struct 结构初始化，包括处理页面的统计信息、初始化信号量（mm->mmap_sem），以及为子进程分配页目录表空间。如果初始化失败，则返回出错信息*/
		goto fail_nomem;

	if (init_new_context(tsk,mm)) /*调用 init_new_context()函数初始化新的上下文，在 i386 平台上，该函数实际上被定义为空（#define init_new_context(tsk,mm) 0)*/
		goto free_pt;

	down_write(&oldmm->mmap_sem);
	retval = dup_mmap(mm); /*调用 dup_mmap()函数，将父进程中所有虚拟内存区域都复制到子进程中来。该函数需要操作父进程 mm_struct 结构的核心数据，因此前后两行使用信号量进行了保护*/
	up_write(&oldmm->mmap_sem);
```

```
    if (retval)
        goto free_pt;   /*检查操作是否成功，如果不成功则返回出错信息*/

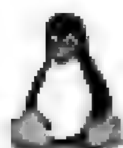
    /*
     * child gets a private LDT (if there was an LDT in the parent)
     */
    copy_segments(tsk, mm); /*调用 copy_segments()函数，复制父进程的 LDT 表，如果
父进程有私有的 LDT 表*/
    /*下面的代码比较简单，不作详细介绍*/
    good_mm:
        tsk->mm = mm;
        tsk->active_mm = mm;
        return 0;

    free_pt:
        mmput(mm);
    fail_nomem:
        return retval;
    }
```

接下来看一下 dup_mmap()函数的实现代码（摘自 linux\linux-2.4.x\kernel\fork.c）：

```
static inline int dup_mmap(struct mm_struct * mm)
  {
      struct vm_area_struct * mpnt, *tmp, **pprev;
      int retval;

      flush_cache_mm(current->mm);   /*flush_cache_mm()函数在 i386 平台上被定义为
空函数*/
      mm->locked_vm = 0;
      mm->mmap = NULL;
      mm->mmap_cache = NULL;
      mm->map_count = 0;
      mm->rss = 0;
      /*在函数 copy_mm()的实现中，已经看到 dup_mmap 的入口参数是新（子）进程的
mm_struct 结构。在程序中首先对该结构进行一些初始化，包括 locked_vm、map_count
等统计信息和 mmap、mmap_cache 等链接信息*/
      mm->CPU_vm_mask = 0;
      mm->swap_address = 0;
      pprev = &mm->mmap;
```




```
/*
 * Add it to the mmlist after the parent.
 * Doing it this way means that we can order the list,
 * and fork() won't mess up the ordering significantly.
 * Add it first so that swapoff can see any swap entries.
 */
spin_lock(&mmlist_lock); /* 由于下面两行将操作全局核心数据结构，因此使用 mmlist_lock 自旋锁进行保护，主要是防止在 SMP 的机器上（一种多处理器机器）出错 */
list_add(&mm->mmlist, &current->mm->mmlist); /*将新进程的 mm_struct 结构加入到 mmlist 链表中，位于父进程的 mm_struct 结构之后*/
mmlist_nr++;  /*将全局变量 mmlist_nr 加 1，以表示地址空间数目的增加*/
spin_unlock(&mmlist_lock); /*与前面的原理相同，使用 mmlist_lock 自旋锁进行保护*/

for (mpnt = current->mm->mmap ; mpnt ; mpnt = mpnt->vm_next) {/*从本行开始，使用 for 循环逐个复制父进程的虚拟内存区域。mpnt 初值为父进程的虚拟内存区域的链头指针 current->mm->mmap，以后每次循环修改为 mpnt = mpnt->vm_next*/
    struct file *file;

    retval = -ENOMEM;
    if(mpnt->vm_flags & VM_DONTCOPY) /*判断当前内存区域标志 vm_flags 是否设置 VM_DONTCOPY，如果是，则该区域不需要复制，直接进入下一个循环*/
        continue;
    tmp = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL); /*为新进程分配一个新的 vm_area_struct 结构（tmp 所指向）*/
    if (!tmp)  /*检查分配是否成功*/
        goto fail_nomem; /*如果不成功，返回出错信息*/
    *tmp = *mpnt; / *直接复制父进程 vm_area_struct 结构（mpnt 所指向），到子进程新创建的 vm_area_struct 结构（tmp 所指向）*/
    tmp->vm_flags &= ~VM_LOCKED;
    tmp->vm_mm = mm;
    tmp->vm_next = NULL; /*更新子进程 vm_area_struct 结构，包括 vm_flags、vm_mm 和
vm_next */
    file = tmp->vm_file;
    if (file) {
        struct inode *inode = file->f_dentry->d_inode;
        get_file(file);
        if (tmp->vm_flags & VM_DENYWRITE)
            atomic_dec(&inode->i_writecount);
```

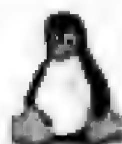
```
            /* insert tmp into the share list, just after mpnt */
            spin_lock(&inode->i_mapping->i_shared_lock);
            if((tmp->vm_next_share = mpnt->vm_next_share) != NULL)
                    mpnt->vm_next_share->vm_pprev_share =
                            &tmp->vm_next_share;
            mpnt->vm_next_share = tmp;
            tmp->vm_pprev_share = &mpnt->vm_next_share;
            spin_unlock(&inode->i_mapping->i_shared_lock);
        } /*检查当前内存区域是否是文件映射区，如果是，则调用 get_file()函数增加
该文件的引用计数。然后，将新的 vm_area_struct 结构加入到共享内存区域链表中*/

        /*
         * Link in the new vma and copy the page table entries:
         * link in first so that swapoff can see swap entries.
         */
        spin_lock(&mm->page_table_lock);
        *pprev = tmp;
        pprev = &tmp->vm_next; /*将新的 vm_area_struct 结构（tmp 所指向）加入到新
（子）进程的虚拟内存区域链表中*/
        mm->map_count++; /*将内存区域数目计数变量 map_count 加 1*/
        retval = copy_page_range(mm, current->mm, tmp);
/*调用 copy_page_range()函数，将当前（父）进程中位于内存区域 tmp 中的页目录、页
表复制到新（子）进程的页目录和页表中。由于此时新进程的页表都还没有创建，因此
该函数逐个检查父进程的页目录项，然后根据页目录项的情况，决定是否需要创建新的
页表。在对页表进行处理时，也需要逐个检查父进程的页表项。一个页表项对应的页面
可能在物理内存中，也可能在映射文件或是交换设备上。另外，由于 Linux 采用了 Copy
On Write 技术，对于父进程中允许写的页面需要特殊处理，即，将该页面在父子进程中
都标记为只读。因此，该函数的实现相当复杂，将在后面介绍*/
        spin_unlock(&mm->page_table_lock);

        if (tmp->vm_ops && tmp->vm_ops->open)
                tmp->vm_ops->open(tmp); /*检查新进程（tmp 所指向）的操作集 ops 是
否定义了 open，如果有，则执行它*/

        if (retval)
                goto fail_nomem;
    }
    retval = 0;
    build_mmap_rb(mm); /*建立 vm_area_struct 结构的链接*/

fail_nomem:
    flush_tlb_mm(current->mm);
```




```
    return retval;
}
```

（2）虚拟地址空间的重建。

fork 系统调用返回后，子进程已经通过虚拟内存复制，创建了自己的地址空间。此后，当进程调用 exec 系统调用，希望执行新的程序时，将根据新的执行映像，为该进程创建新的虚拟地址空间。通常将这个过程叫做虚拟地址空间的重建。

在传统的实现中，可执行映像从磁盘上读入，并全部装入到进程的虚拟地址空间，显然这是一个相当耗时的操作。另外，由于程序运行的局部性原理，映像的某些部分可能根本就用不到。

为了提高系统的效率，在 Linux 中（该方法最初来自 System V）采用内存映射机制来处理映像文件的装入。在 vm_area_struct 结构中，由一个 file 结构的 vm_file 域和一个无符号长整型类型的 vm_pgoff 域分别表示该内存区域映射文件的文件指针和偏移值。实际上内存映射机制正是使用这两个域描述了某段内存空间对应的内容在文件中的位置。这样，在重建虚拟地址空间时，只需建立一系列的数据结构，描述某段内存区域内容在可执行映像中的位置，只有当进程真正使用该区域时，才将其装入内存。使用这种机制，就避免了将映像中并不使用的部分也装入了内存中。

当可执行映像映射到进程虚拟地址空间时，将产生一组 vm_area_struct 结构来描述可执行映像内容与虚拟地址区域的对应关系。每个 vm_area_struct 结构描述可执行映像的一部分，可以是代码段，也可以是数据段或 BSS（未初始化的数据）段等。该部分功能实际上由 do_mmap_pgoff()函数完成（由 exec()函数经多次调用进入该函数），下面来看看这部分的代码。

函数 do_mmap_pgoff()定义如下：

unsigned long do_mmap_pgoff(struct file * file, unsigned long addr, unsigned long len,
        unsigned long prot, unsigned long flags, unsigned long pgoff)

其中，参数 file 是需要建立映射的文件；addr 为虚拟地址的开始位置；len 为映射的长度；prot 为保护标志；flags 为映射标志；pgoff 为文件映射区域的偏移值。

该函数比较复杂，来看一下它的代码（摘自 linux\linux-2.4.x\mm\mmap.c,部分）：

```
unsigned long do_mmap_pgoff(struct file * file, unsigned long addr, unsigned long len,
        unsigned long prot, unsigned long flags, unsigned long pgoff)/*函数的声明部分*/
    {
        struct mm_struct * mm = current->mm;

        ......（参数合法性检查）
        /* Obtain the address to map to. we verify (or select) it and ensure
         * that it represents a valid section of the address space.
         */
        addr = get_unmapped_area(file, addr, len, pgoff, flags);/*调用 get_unmapped_area()函
```

```
数，根据传入的希望映射的地址和标志等找出实际可用的映射地址。如果在 flags 中指定
了映射类型为 MAP_FIXED，则开始地址就是 addr，并且要求 addr 是页对齐的。如果 addr
为 0，则从 1GB 以后的地址寻找一个尚未映射的，大小超过 len 的内存区域，并返回它
的首地址*/
    if (addr & ~PAGE_MASK) /*检查该地址是否页对齐，如果不是，返回出错信息*/
        return addr;

        ...（操作权限检查）
    /* Clear old maps */
    /*下面处理老的虚拟内存映射的清除工作*/
  munmap_back:
    vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);/* 调 用
find_vma_prepare()函数，根据 addr 参数找到需要清除的第一个虚拟内存区域*/
    if (vma && vma->vm_start < addr + len) {/*判断即将映射的地址是否和找到的虚拟
内存区域冲突，即判断将要映射的地址正在找到的虚拟内存区域中*/
    if (do_munmap(mm, addr, len))   /*如果存在冲突，就清除老的映射*/
        return -ENOMEM;
    goto munmap_back; /*使用 goto 跳转到 munmap_back，继续下一个冲突虚拟内存区
域的处理*/
    }
        ...（检查地址空间限制，是否可写、可扩展等）
    /* Determine the object being mapped and call the appropriate
     * specific mapper. the address has already been validated, but
     * not unmapped, but the maps are removed from the list.
     */
    vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);   /*为即将创建的
vm_area_struct 结构分配内存空间*/
    if (!vma)
        return -ENOMEM;   /*检查分配是否成功，如果不成功，则返回错误信息*/

    vma->vm_mm = mm;
    vma->vm_start = addr;
    vma->vm_end = addr + len;
    vma->vm_flags = vm_flags;
    vma->vm_page_prot = protection_map[vm_flags & 0x0f];
    vma->vm_ops = NULL;
    vma->vm_pgoff = pgoff;
    vma->vm_file = NULL;
    vma->vm_private_data = NULL;
    vma->vm_raend = 0;
    /*对新分配的 vm_area_struct 结构进行一些初始化，主要是填写一些关键的字段，
包括 vm_mm、vm_start、vm_end、vm_page_prot、vm_flags 等*/
```




```
    if (file) {
            ...（处理文件映射）
    }
/*对该区域映射文件的情况进行一些处理。其中，包括读写权限、增长方向、是否共享等，并调用其操作集中的 mmap()函数(error = file->f_op->mmap(file, vma);) */
    /* Can addr have changed??
     *
     * Answer: Yes, several device drivers can do it in their
     *         f_op->mmap method. -DaveM
     */
    if (addr != vma->vm_start) {
            ...（处理地址变化的情况）
    }

    vma_link(mm, vma, prev, rb_link, rb_parent); /*调用 vma_link()函数，将新创建的vm_area_struct 结构加入到链接数据结构中*/
        /*下面是一些扫尾代码和出错处理代码*/
    if (correct_wcount)
        atomic_inc(&file->f_dentry->d_inode->i_writecount);

out:
    mm->total_vm += len >> PAGE_SHIFT;
    if (vm_flags & VM_LOCKED) {
        mm->locked_vm += len >> PAGE_SHIFT;
        make_pages_present(addr, addr + len);
    }
    return addr;

        ...（出错处理）
}
```

到此为止，一个 vm_area_struct 结构的创建和初始化工作就完成了。上层代码反复调用 do_mmap_pgoff()函数，将可执行映像的各部分内容全部映射到进程的虚拟地址空间。这样虚拟地址空间的重建就完成了。

5. 缺页处理

由虚拟地址的重建过程可以看出，可执行程序的内容并没有全部装入到物理内存中。当程序访问那些尚未装入物理内存的虚拟内存区域时，程序的运行将产生异常，如在 i386 平台上，处理器将触发一个缺页异常（page fault）。另外，在程序运行过程中，当程序访问已被换出的页面时，也会引起缺页异常。上面两种情况都是可以预期到的“正常出错”，还有一些情况，比如程序编写出错，访问了受保护的地址空间，或是没有操作权限的地址，

同样会引起缺页异常。

Linux 使用 page_fault 中断处理程序来处理页面异常。而具体的处理工作，实际上调用了 do_page_fault()函数来完成。

该函数首先定位发生异常的虚拟地址，然后在发生异常的进程的所有虚拟内存区域 vm_area_struct 结构中查找包含该地址的区域。如果找不到对应的 vm_area_struct，或是该 vm_area_struct 的权限设置禁止该访问，则向该进程发送 SIGSEGV 信号；如果找到了满足条件的 vm_area_struct 结构，再查找其页表项，根据页表项的内容，决定其处理办法，包括读入页面内容、处理 Copy On Write 页面和建立一个匿名页等。

当异常处理完返回后，需要的页面已经装入内存，这时异常处理程序将返回，原来的进程就可以继续执行下去。

以下是 do_page_fault()函数的实现代码(摘自 linux\linux-2.4.x\arch\i386\mm\fault.c 部分)：

```
asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    struct task_struct tsk;/*发生缺页的任务*/
    struct mm_struct mm; /*发生缺页的内存空间*/
    struct vm_area_struct * vma; /*发生缺页的虚拟地址空间*/
    unsigned long address; /*发生缺页的地址*/
    unsigned long page;
    unsigned long fixup;
    int write;
    siginfo_t info;

    /* get the address */
    __asm__("movl %%cr2,%0":"=r" (address));     /*取页面错误的线性地址。在 i386
平台上，每次发生页面访问错误时，CPU 硬件就把发生页面错误的线性地址放到 cr2 寄
存器中，该语句即是将 cr2 寄存器中的内容复制到 address 变量中*/
    ...
    down_read(&mm->mmap_sem);/*申请使用 mm->mmap_sem 信号量，保证后面的操
作互斥进行*/

    vma = find_vma(mm, address);/*调用函数 find_vma()在当前进程中查找 vm_end 大
于 address 的第一个 vm_area_struct 结构*/
    if (!vma)/*检查是否找到这样的 vma，如果找不到，则该次虚拟地址访问非法，将
跳转到 bad_area 处理*/
        goto bad_area;
    if (vma->vm_start <= address)/*判断如果 vma->vm_start <= address，则表明该次访
问的虚拟地址是在一个 vm_area_struct 结构中，产生页面错误的原因应该是其页面尚未
调入内存，跳转到 good_area 进行处理*/
        goto good_area;
    if (!(vma->vm_flags & VM_GROWSDOWN))
```




```
/*是 vma->vm_start > address 的情况，即该地址在一个 vm_area_struct 结构的前面。检查该 vm_area_struct 结构是否是向下增长（堆栈是向下增长的）的，如果不是，就转到 bad_area 处理*/
            goto bad_area;
    if (error_code & 4) {/*检查该次异常是否由压栈操作产生，如果是，那么引发异常的地址应该在当前栈顶 （regs->esp）附近*/

            if (address + 32 < regs->esp)/*检查 address（引发异常的地址）是否偏离当前栈顶太远，如果是的话，就跳转到 bad_area 处理*/
                    goto bad_area;
    }
    if (expand_stack(vma, address))/*扩展堆栈，因为程序能执行到此处，必然是因为堆栈空间不足引起的，如果扩展失败，将跳转到 bad_area 处理*/
            goto bad_area;
/*
 * Ok, we have a good vm_area for this memory access, so
 * we can handle it..
 */
good_area:/*程序能执行到此处，说明该 address 在合法的 vma 内*/
    info.si_code = SEGV_ACCERR;
    write = 0;
    switch (error_code & 3) {/*根据异常的类型进行处理，异常类型由 error_code 的低两位表示*/
            default:  /* 3: write, present */
#ifdef TEST_VERIFY_AREA
                    if (regs->cs == KERNEL_CS)
                            printk("WP fault at %08lx\n", regs->eip);
#endif
                    /* fall through */
            case 2:        /*异常类型 2 表示写一个不存在的页面*/
                    if (!(vma->vm_flags & VM_WRITE))/*检查如果当前 vma 区域是只读的，则权限操作错误，跳转到 bad_area 处理*/
                            goto bad_area;
                    write++;
                    break;
            case 1:  /*异常类型 1 表示读一个存在的页面，如果是该种情况，应该不会引发异常，因此，出现这种情况必然是有问题，接下来转到 bad_area 处理*/
                    goto bad_area;
            case 0:        /*异常类型 0 表示读一个不存在的页面*/
                    if (!(vma->vm_flags & (VM_READ | VM_EXEC)))/*检查如果当前 vma 区域不是可读和可执行的，则权限操作错误，跳转到 bad_area 处理*/
                            goto bad_area;
```

```
    }

    survive:/*已经能够断定，当前需要进行正常的缺页处理*/
    /*
     * If for any reason at all we couldn't handle the fault,
     * make sure we exit gracefully rather than endlessly redo
     * the fault.
     */
    switch (handle_mm_fault(mm, vma, address, write))
    {                                /*调用 handle_mm_fault()函数，该函数处理正常的缺页。该函数首先定位发生异常的地址(address)所在的页目录（pgd），然后分配一个新的页表，接着调用 handle_pte_fault()函数，在该函数中又进一步调用了 do_no_page 和 do_swap_page 函数，分别处理页面尚未装入和页面在交换设备上的情况。当handle_mm_fault()函数执行完成后，所需要的页面就已经调入内存了*/
        case 1:
            tsk->min_flt++;
            break;
        case 2:
            tsk->maj_flt++;
            break;
        case 0:
            goto do_sigbus;
        default:
            goto out_of_memory;
        }

        ...

        up_read(&mm->mmap_sem);
        return;

        ...

    }
```

到此为止，缺页处理的主要过程就分析完成了，后面的代码主要是错误情况的处理和扫尾代码，就不再分析了。

5.3 小结

存储器管理是操作系统的基本服务之一，存储器管理负责合理地分配内存空间，实现

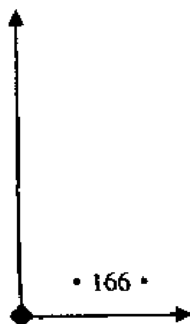


存储保护、内存共享和虚拟存储器管理等。

主要的存储器管理方案有：分区管理、段式管理、页式管理、段页式管理。每一种存储管理方案都要从内存空间的划分、用户程序的划分、逻辑地址形式、内存分配方式、数据结构设置、必要的硬件支持、地址映射过程以及如何提高效率等多方面进行设计与实现。虽然不同存储管理方式实现存储管理的方法有所不同，但它们都要有相应的硬件作支撑。

5.4 思考题

1. 早期存储器管理有哪几种方法？各自的优缺点是什么？
2. 什么是离散内存分配？
3. 虚拟存储器实现的难点是什么？
4. 请求分页系统中，页表的主要内容及其作用是什么？
5. 联想存储器 TLB 的作用，实现 TLB 应该有哪些问题需要考虑？
6. 段式和页式存储管理是如何实现存储保护的？
7. Linxu 内存管理的主要特点是什么？



第6章 中断处理

知识点：

- 中断的概念和中断类型
- 中断服务程序与中断向量表、中断描述符表
- 中断响应过程
- 中断服务程序设计

本章导读：

中断是计算机中一个很重要的概念，所谓"中断"就是由微处理器的内部硬件或外部设备引起的，暂停当前程序执行，转而去执行一个特殊处理程序，并在执行完后，恢复原来程序的执行过程。中断是为克服I/O接口程序采用查询方式实现所带来的处理器低效率的缺点而引入的。中断技术是现代计算机发展的一种重要技术，计算机通过中断控制方式来实现计算机与外围设备之间及计算机内部部件之间的数据交换。



6.1 中断概述

中断是微处理器处理随机事件的一种机制。所谓中断，就是当 CPU 正常运行程序时，由于随机的事件（包括内部事件和外部请求）引起 CPU 暂停正在运行的程序，转去执行某个特定的程序，并在执行后返回原来被暂停的程序处继续向下执行的过程。

引入中断主要是为了解决 CPU 与外部设备间在速度方面不匹配的问题。通常，计算机可以通过轮询方式来实现各种数据的输入或输出，但是通过轮询方式实现数据的输入或输出，会使 CPU 浪费很多的时间去等待外部设备（特别是低速外部设备）提出传输请求，从而降低 CPU 的使用效率。在各种微型计算机系统中，常常利用 CPU 的中断机制来完成与外部设备间的数据传送，以期用最少的响应时间来处理所有外部设备的服务请求，从而提高整个计算机系统的性能。

6.1.1 中断源

所谓中断源，是指发出中断申请的外部设备或引起中断的内部原因。在 80x86 系列的系统中，它的中断源既可以来自 CPU 内部，也可以来自 CPU 的外部（即外部的接口芯片或外部设备）。通常把因外部请求而引发的中断叫做外部中断或硬件中断，把因为处理器内部运行异常而引发的中断叫做内部中断。

外部中断是由外部设备确定的硬件中断，其中断源是外部接口设备，它是微机系统与外部设备进行数据交换和信息交换的最主要途径之一。外部中断分为可屏蔽中断和非可屏蔽中断两种。

80x86 系列的 CPU，其外部中断仅有两个引脚：可屏蔽中断引脚 INTR 和不可屏蔽中断引脚 NMI。通过 INTR 引脚向 CPU 提出请求的中断称为可屏蔽中断，它接受 IF 标志位（CPU 标志寄存器中的控制位）的影响和控制。当 IF 被软件（即 STI 指令）置 1 时，表明可屏蔽中断被允许，CPU 可以响应中断；当 IF 被软件（即 CLI 指令）置 0 时，表明中断被禁止响应，CPU 此时不响应可屏蔽中断。在 80x86 系统中，可屏蔽中断源产生的中断请求信号，通常都通过 8259 中断控制器进行优先权判优后，由 8259 向 CPU 传送中断请求信号 INTR 和中断类型号。

通过 NMI 引脚向 CPU 提出请求的中断称为非可屏蔽中断，它是不能被 IF 标志禁止的中断。通常用于处理紧急事件，如电源掉电等。非可屏蔽中断源产生的中断请求信号直接送到 CPU 的非可屏蔽中断引脚 NMI。

内部中断也分两类，一类是在程序运行中的特殊事件（如除数为零、运算溢出或单步跟踪及断点设置等）引发的中断，称为内部硬件中断；另一类是由软中断指令 INT n 引起的中断，称为软件中断。所有的内部中断都是非屏蔽的。

6.1.2 中断类型号、中断向量表和中断描述符表

80x86 系列处理器的中断系统能够处理 256 个不同的中断源，每个中断源都有一个对应的中断处理程序。为了能区分不同的中断，每个中断源都被赋予惟一的编号，叫做中断类型号，又叫中断号。在 80x86 系统中，共有 256 个中断号（0~255）。中断所对应的中断向量通过几种不同的方式来得到：异常是由内部（80386/80486 内部）根据异常的类型设置不同的中断号；软件 INT n 指令中包含了中断号 n；可屏蔽硬件中断是由引起中断的外部设备通过总线的中断响应序列来得到 8 位的中断向量；而非屏蔽硬件中断的中断号被指定为 2。Intel 在设计处理器时，保留了前 32 个中断号，0H~1FH 供处理器内部使用，其后的 20H~FFH 供用户使用。保留的中断一般用于实现系统功能。

在 Intel 的高档处理器（80386 以上）中，有保护模式和实模式之分。而且，其对应的中断的处理过程也有所不同。在实模式下，通常把中断服务程序的入口地址（段地址：偏移量）叫做中断向量。因为存在 256 个中断，所以就有 256 个中断向量。把 256 个中断向量存放在一起，形成一张表，即得到中断向量表。在该表中，每个中断向量占用 4 个字节（高地址 2 个字节是段基地址，低地址 2 个字节是偏移量），256 个中断向量共占用 1KB 地址空间，位于内存的最低端 0000H~03FFH。

在保护模式下，中断服务程序的入口地址不再用中断向量来表示。为了便于中断管理，从 80286 开始，Intel 系列微处理器中引入了中断描述符和中断描述符表（IDT）的概念。

中断描述符表由多个中断描述符组成，它描述了与某中断对应的中断服务程序的入口地址及其管理信息，包括段选择子和偏移量等。在 i386 平台上，每个中断描述符占用 8 个字节，其格式如图 6-1 所示。

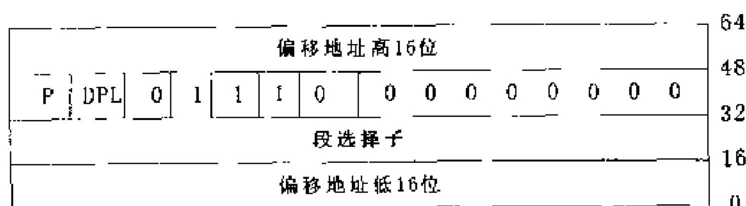


图 6-1　i386 平台上中断描述符占用示意图

所有的中断描述符在内存中连续存放就构成中断描述符表（Interrupt Descriptor Table）。该表最多能容纳 256 项中断描述符，每项占用 8 个字节，一共需要 2KB 的内存空间。与实模式不同的是，该段内存空间可以在内存中浮动，系统中由一个中断描述符表寄存器（IDTR）专门指向该表首地址。

6.1.3 中断服务程序及其入口地址

每次系统产生中断后，都会转向一个特定的子程序执行，通常把这个程序叫做中断服



务程序。中断服务程序根据不同的中断号和引发中断的原因进行处理。中断服务程序往往用于实现一些与异步输入/输出相关的系统功能。

中断服务程序一般由 4 部分组成：保护现场、中断服务代码、恢复现场、中断返回。所谓保护现场，是因为寄存器可能存放了主程序执行时的有用内容，为了返回后不破坏主程序在断点处的状态，应将有关寄存器的内容压入堆栈。当然，中断服务程序用不到的寄存器不必保护。恢复现场是指中断服务程序完成后，把压入堆栈的相关寄存器的内容再弹回寄存器中，即使 CPU 的有关状态恢复到被中断之前。有了保护现场和恢复现场的操作，就可以保证在返回断点后，能正确无误地继续执行。

要执行中断服务程序，必须知道相应的入口地址。在 80x86 系列的处理器中，实模式和保护模式下的中断服务程序的入口地址计算方法有所不同。

在实模式下，每个中断源占用中断向量表的 4 个字节单元，存放其服务程序入口地址，即中断向量。其中高 2 字节表示中断服务程序入口地址的段基址 CS，而低 2 字节表示中断服务程序入口地址的偏移量。在中断向量表中，中断向量存放的地址＝中断类型号 n×4。

在保护模式下，中断服务程序的入口地址是通过描述符结构来存储的。其查找过程可以分成如下 3 个步骤：

（1）根据中断类型号在中断描述符表中查找中断描述符。中断描述符表的起始地址存放在 IDTR 寄存器中。中断描述符在内存中的起始地址＝中断描述符表起始地址+中断类型号 n×8。

（2）根据中断描述符中的选择子中的 TI 字段在 GDT 或 LDT 查找段描述符的起始位置（当 TI＝0，在 GDT 中查找；当 TI＝1，在 LDT 中查找）。段描述符的起始位置＝描述符表的起始地址（GDT 表起始地址存放在 GDTR 寄存器中，LDT 表起始地址存放在 LDTR 寄存器中）+索引值（选择子高 13 位）×8。

（3）将段描述符中的段基地址与中断描述符中的偏移值相加即可得到中断服务程序的入口地址。

6.1.4 中断优先级和中断嵌套

当系统中有几个中断同时发生时，微处理器必须确定对它们的响应次序。通常根据中断源提出中断申请的轻重缓急为每个中断源确定其优先权。微处理器将按照每个设备优先权的高低来提供中断服务，优先级高的先执行。在分配优先权时，一般是按照该中断的紧急和重要程度来安排的。在 80x86 系列系统中，默认的中断源的优先权由高到低的顺序依次为：

内部异常中断（除单步）→INT n→NMI→INTR→单步中断

如果当 CPU 在处理优先级较低的中断时，能够响应更高优先级的中断请求，则该系统有中断嵌套处理能力。中断嵌套系统能够更好地响应外部紧急事件的请求，但是中断嵌套的实现往往增加了操作系统实现的难度。图 6-2 显示了两级中断嵌套的情况。

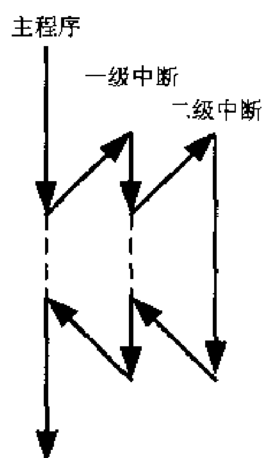


图 6-2　两级中断嵌套示意

6.2　中断机制

6.2.1　中断响应过程

从中断源提出请求中断开始，到中断请求处理完毕后返回的全过程，叫做中断过程。它可分为中断请求、中断判优、中断响应、中断服务和中断返回 5 个阶段。接下来对这 5 阶段分别加以介绍。

1. 中断请求

引起中断的原因很多。当外部设备需要 CPU 进行控制，或者 CPU 运行出现异常情况时，都要发出中断请求。

外部设备发起中断请求的原因很多，诸如字符 I/O 设备等待读取或者已经接收到了一个字符；块设备的读写操作已经完成；外部设备故障、传输错误、定时器时间已到等都可以是中断源。从主机内部来说，系统掉电、硬件故障、软件错误、设置断点或单步执行操作等也可以是中断源。这些中断源的共同特点是都需要 CPU 对其进行适当的处理。

外部设备接口提出的中断请求按照请求信号的不同分为边沿请求和电平请求。中断请求信号由低到高或由高到低的跳变是边沿触发的请求，中断请求信号持续为高电平或低电平是电平触发的请求。为了保证出现的每一事件均能得到 CPU 处理，中断请求机制应保持或记录该事件，直到开始处理或已被处理完为止。

2. 中断判优

由于系统往往具有多个中断源，加之中断请求的随机性，可能出现两个或两个以上的中断源同时发出中断请求的情况。这就要求 CPU 能够判别优先级最高的中断源，并对其加以响应。中断判优就是要解决请求中断的事件优先级的顺序问题。中断判优的方法分为软



件和硬件两种。现在被广泛使用的中断控制器芯片 82C59 就提供了硬件判优支持。

3. 中断响应

中断响应就是 CPU 暂停现在正在进行的处理任务，转向处理与中断请求相对应的处理程序的过程。中断响应过程应解决如下问题：首先要确定当前请求中断的中断源，以便执行对应的中断处理程序；其次，在执行中断处理程序之前，还必须进行现场保护。

对于非可屏蔽中断，其中断向量号约定为 2。对于可屏蔽中断，在响应中断后，CPU 会启动中断识别周期，由硬件完成中断向量号的识别工作。可屏蔽中断的识别过程如下：

（1）CPU 接收到中断控制器（8259）的中断请求（INTR）后，在当前指令执行完成后，进入中断响应周期。

（2）在中断响应周期，CPU 向中断控制器发出两个响应脉冲 INTA，第一个响应脉冲通知中断控制器，CPU 已经响应外部中断，中断控制器应该将中断向量号放到数据总线上（D0-D7）。在第二个响应脉冲到来时，CPU 读取中断类型号。

（3）保存现场，将标志寄存器压栈保存，清除 IF 和 TF 位，以禁止再次响应中断和陷阱。

（4）将当前执行指令的下一条指令的地址压入堆栈，以便在执行完中断后返回继续执行。

（5）CPU 根据第 2 步读取到的中断向量号，计算中断服务程序的入口地址，并转入该地址执行。

4. 执行中断服务程序

在前面的过程中，CPU 已经确定了中断服务程序的入口地址，接下来就是转入中断服务程序执行。中断服务程序可以由系统提供或是用户编写。无论是谁提供，中断服务程序中应进行的基本操作步骤如下：

（1）保护现场。在 80x86 处理器中，CPU 自动保存相应中断时的标志寄存器内容和断点地址。用户只需要保存在主程序和中断服务程序中均被使用到的寄存器。保护现场普遍使用的方法是将需要保存的寄存器内容压入堆栈。

（2）开中断。在中断响应过程中，CPU 自动清除了 IF 位和 TF 位，目的是暂时禁止中断，以保护中断识别周期不会被新的中断所打断。在进入中断服务程序后，可以根据需要决定是否需要打开中断。其原则是，如果在处理该中断的过程中，需要响应更高级别的中断（即前面提到的中断嵌套），则必须开中断；反之，则不需要开中断。

（3）执行中断处理代码。中断服务程序最重要的部分是中断处理代码，该段代码将完成中断处理的核心功能，诸如执行输入/输出操作或是非常事件的处理等。

（4）关闭中断。如果在第 2 步中已经打开了中断操作，则在该处就应该进行关闭中断操作，以保证中断返回后主程序的正常执行。

（5）恢复现场。在中断返回之前，必须将机器的状态恢复到中断时的状态，以保证主程序正常执行。恢复现场普遍使用的方法是将原来压入堆栈的寄存器内容弹出堆栈，其顺序与压入时相反。

（6）中断返回。中断服务程序的最后一条指令是中断返回指令（在 80x86 平台上是 iret）。

执行该指令后，将自动恢复标志寄存器的值，并从堆栈中取得返回地址。同时，跳转该处执行。

6.2.2 中断服务程序设计

中断服务程序的编写与一般程序有所不同，这里以其与一般的 C 语言子函数的对比为例，来说明其编写的特殊之处：

- 中断服务程序是在 CPU 收到中断请求后，自动执行；而 C 语言的子函数需要用户显示调用才能执行。
- 中断服务程序的执行具有随机性，其执行由外部中断和内部异常等触发；而 C 语言的子函数是由用户显示调用，其执行的时间可以预测。
- 中断服务程序没有参数。它与其他程序通信，只能通过全局变量或是共享内存进行；而 C 语言的子函数可以使用参数和返回值。
- 中断服务程序必须短小简练。由于中断服务程序将打断主程序的执行，因此，中断服务程序不能长时间占有 CPU，否则，将影响主程序的正常执行，特别是与时间相关的功能；而 C 语言的子函数没有这个限制。
- 中断服务程序的编写必须包含特定的内容，包括保存恢复现场和开关中断等；而 C 语言的子函数没有这个要求。

下面是一个简单的时钟中断服务程序的代码：

```
    .globl Ttimer_val
    .globl timerisr
timerisr:
    pushl    eax            #保存现场，因为在中断服务程序中将使用 eax 寄存器，
                             故将 eax 中的内容保存到堆栈中
    incl Ttimer_val         #将时钟计数变量 Ttimer_val 加 1
    movb    $0x20, al
    outb al, $0x20          #输出 EOI（中断结束信号）到 8259
    popl eax                #恢复现场
    iret                    #中断返回
```

6.3 使用中断驱动串口

6.3.1 PC 机串口的基本概念

计算机与外部设备间的通信有两种方式：并行通信和串行通信。串行通信降低了架设传输线路的开销，因而使通信成本降低，尤其是在远程传送时更为显著。但因增加了数据格式的串-并、并-串转换，位计数及传送定时控制技术等，使得串行通信比并行通信更为复



杂。另外，当传输线的传输速度一定时，串行通信比并行通信的速度要低。但总的来说，串行通信在远程通信和低速传输数据时，有相对的优势。

目前，PC 机中用于通用异步信息传输的串口安装在 UART（Universal Asynchronous Receiver Transmitter 通用异步收发器）芯片旁边。早期使用的 UART 的型号为 8250 或 16450，这两种型号的芯片都不能满足当今快速 Modem 的需要。目前，PC 机普遍使用的是 16550 型号的 UART，最新型的 UART 的型号是 16650 和 16750，可以满足 ISDN 或更高速的连接的需要。

16550 是美国国家半导体公司（National Semiconductor Inc.）推出的、与 Intel 微处理器兼容的、使用非常广泛的 UART。它是一种可以连接任何类型虚拟串行接口的可编程通信接口。16550 可用于输入、输出数据，并具有一个可编程波特率发生器，可以有效地减轻微处理器的负担。

16550 有两种封装形式，一种是 40 引脚的 DIP，另一种为 44 引脚的 PLCC。16550 的一个主要特性是它的内部兼有接收器和用于发送的 FIFO 存储器。每个 FIFO 存储器均为 16 字节，UART 在接收到 16 字节的数据之后才通知微处理器，或在微处理器必须等待发送之前保持 16 字节的数据量。FIFO 使得这种 UART 非常适合连接到高速系统上，因为它减少了 CPU 处理 UART 的时间。表 6-1 列出了 16550 芯片的主要功能引脚。

表 6-1　16550 主要引脚说明

A_2~A_0	地址输入，用于选择进行编程和数据传送的内部寄存器
ADS #	地址选通输入，用于锁存地址线和片选线。Intel 系列处理器不需要，此引脚接地
BAUDOUT	波特率输出引脚，是由发送器的波特率发生器产生的时钟信号。常与 RCLK 输入连接，产生与发送器时钟相等的接收器时钟
CS_0、CS_1、CS_2	片选输入，3 个引脚均有效才能选中 16550 UART
D_7~D_0	数据总线
DSR	数据装置准备就绪，输入引脚
DTR	数据终端（16550）准备就绪，输出引脚
INTR	中断请求信号，连接到 8259。当发生接收错误、接收缓冲已满或是发送缓冲为空时申请中断
MR	复位信号，用于初始化 16550，与系统的 REST 信号相连
RCLK	接收器时钟信号，输入引脚
RXRDY	接收器就绪信号
TXRDY	发送器就绪信号
WR	写操作信号
SIN、SOUT	串行数据引脚，SIN 为接收串行数据，SOUT 为发送串行数据

按照逻辑功能，16550 可以被分为以下几个部分：

- 总线连接逻辑：该逻辑包括 I/O 数据总线、地址线、读写信号线等，通过总线可直接与 CPU 实现连接。

- 芯片选择逻辑：16550有3个片选信号CS_0、CS_1 、CS_2，只有在CS_0、CS_1都为1，且CS_2为0时，芯片才被选中，且在ADS为0时，片选信号才能生效。
- MODEN 控制逻辑：16550 所具有的 Modem 控制逻辑主要是指 DTR/DSR、RTS/CTS、RI/DCD这3对握手信号及其应答逻辑。这些信号的相互配合可以实现MODEN与16550之间可靠的数据传输。
- 串行数据收发逻辑：收发逻辑包括SIN、SOUT和接收时钟电路。当UART接收数据时，在接收时钟信号RCLK的作用下，将Sin输入端上的数据串行移入接收缓冲寄存器RBR，供CPU读取；当UART发送数据时，在发送时钟信号作用下，将数据总线上的数据写入到发送寄存器THR中，再从Sout端上串行输出。
- 中断请求逻辑：16550 内部定义了 4 级中断，无论哪一级中断产生请求，均通过INTR引脚发出中断信号。

16550在PC机上映射到两个串口上，其起始端口地址分别是0x3F8（COM1）和0x2F8（COM2）。每个COM口都对应16550内部的10个8位寄存器和一个16位的波特率发生寄存器。其端口地址和简写等如表6-2所示。

表 6-2 PC 机串口地址表

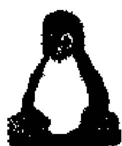
寄存器信息 / 寄存器	地址信号			简写	串口1寄存器地址	串口2寄存器地址	备注
	A_2	A_1	A_0				
接收缓冲寄存器（读）	0	0	0	RBR	0X3F8	0x2F8	DL位为0时
发送保持寄存器（写）	0	0	0	THR	0X3F8	0x2F8	DL位为0时
中断使能寄存器	0	0	1	IER	0X3F9	0X2F9	DL位为0时
中断标识寄存器（读）	0	1	0	IIR	0X3FA	0X2FA	
FIFO控制寄存器（写）	0	1	0	FCR	0X3FA	0X2FA	
线路控制寄存器	0	1	1	LCR	0X3FB	0X2FB	
调制解调器控制寄存器	1	0	0	MCR	0X3FC	0X2FC	
线路状态寄存器	1	0	1	LSR	0X3FD	0X2FD	
调制解调器状态寄存器	1	1	0	MSR	0X3FE	0X2FE	
临时寄存器	1	1	1	SCR	0X3FF	0X2FF	
波特率除数锁存寄存器（低字节）	0	0	0	LSB	0X3F8	0x2F8	DL位为1时
波特率除数锁存寄存器（高字节）	0	0	1	MSB	0X3F9	0X2F9	DL位为1时

注：DL位代表线路控制寄存器LCR的最高位。

其各寄存器的含义如下：

- 接收缓冲寄存器（RBR）与发送保持寄存器（THR）

这两个寄存器共用同一个地址，对UART的读操作可将数据从RBR中读出，对UART的写操作可将数据从数据总线写入THR。



- 波特率除数锁存寄存器 LSB、MSB

该寄存器为 16 位寄存器，由于 16550 的数据总线只有 8 位，因此必须分为两次来操作。除数的值可由 UART 的工作频率和波特率共同确定，计算公式为：

除数值= UART 的工作频率/（16×期望的波特率）

其中 16550 的工作频率为 1.8432×10^6Hz。访问该寄存器时，应该设置 LCR 的最高位，然后通过 000、001 地址来访问。

- 中断使能寄存器 IER

16550 有 4 种类型的中断，分别以 IER 的低 4 位来表示，每一位对应一个中断源，当相应位为 1 时表示该中断有效。而 IER 的高 4 位没有定义，一般将其置为 0。该寄存器各位含义如下：

  - IER.0：表示接收中断。
  - IER.1：表示发送中断。
  - IER.2：表示接收线路状态中断。
  - IER.3：表示调制解调器中断。
  - IER.4~ IER.7 未定义。

- 中断标识寄存器 IIR

由于 16550 中断请求信号只有一个，无论哪个中断源产生了中断，中断请求信号都会生效。为了正确识别中断源，以便转入相应的中断服务子程序，就必须从 IIR 中读出中断识别字，据此来判断中断源的类型。IIR.0＝0 表示有中断产生；IIR.1~ IIR.3 定义中断源的类型；IIR.6~ IIR.7 定义接收和发送模式，当 IIR.6、7=11 时，UART 定义为 FIFO 模式，当 IIR.6、7=00 时，UART 定义为 DMA 模式。需要注意的是，IIR 寄存器是只读的。

- 线路控制寄存器 LCR

线路控制寄存器用于设置通信数据格式，即一个数据帧中的数据位数、停止位数、奇偶校验方式等。图 6-3 给出了 LCR 每一位的含义。

- 线路状态寄存器 LSR

线路状态寄存器提供串行数据传送和接收时的状态，供 CPU 判断系统状态之用。该寄存器有一个特点，就是在读取相关数据寄存器时，该寄存器的相应位自动清零，而数据寄存器的内容发生变化时，该寄存器的相应的位置为 1。该寄存器各位含义如下：

  - LSR.0=1：接收数据寄存器已收到数据。
  - LSR.1=1：出现越位错误。
  - LSR.2=1：出现奇偶校验错。
  - LSR.3=1：出现帧格式错。
  - LSR.4=1：RBF 检测到空状态已持续一个完整帧传输时间。
  - LSR.5=1：发送保持寄存器空。
  - LSR.6=1：发送移位寄存器空。
  - LSR.7=1：有奇偶校验错误，或帧格式错误，或空号错误出现。

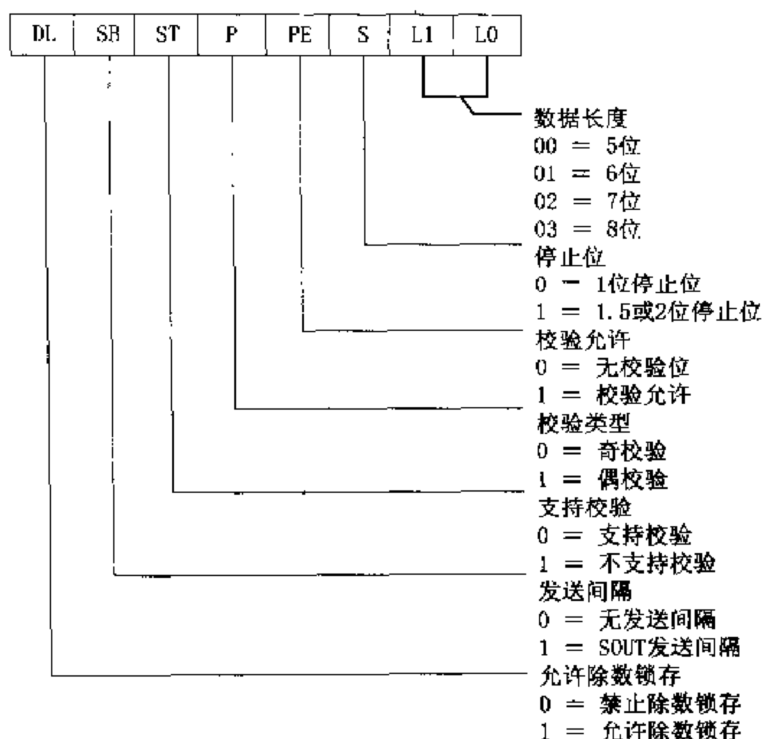


图 6-3 线路控制寄存器 LCR 的数据格式

- 调制解调器控制寄存器 MCR

MCR 用于提供 UART 和 Modem 之间会话所需的握手信号，它与 MSR 配合，可以实现 UART 和 Modem 之间的完整交互。该寄存器各位含义如下：

➢ MCR.0=1：DTR 管脚信号有效，反之无效。
➢ MCR.1=1：RTS 管脚信号有效，反之无效。
➢ MCR.2=1：OUT1 管脚信号有效，反之无效。
➢ MCR.3=1：OUT2 管脚信号有效，反之无效。
➢ MCR.4=1：设置 16550 进入自测试诊断模式，MCR 和 MSR 构成一个自行闭合的回路，反之无用。
➢ 位 MCR.5~ MCR 7 未定义。

- 调制解调器状态寄存器 MSR

该寄存器提供调制解调器的状态标志，它与 MCR 配合使用，可实现 UART 和 Modem 之间的完整交互。该寄存器各位含义如下：

➢ MSR.0=1：CTS 信号已变化。
➢ MSR.1=1：DSR 信号已变化。
➢ MSR.2=1：RI 信号已变化。
➢ MSR.3=1：DCD 信号已变化。
➢ MSR.4=1：CTS 信号有效。
➢ MSR.5=1：DSR 信号有效。



- MSR.6=1：RI 信号有效。
- MSR.7=1：DCD 信号有效。

6.3.2 PC 机串口驱动程序的实现

下面，来看一个 PC 机串口驱动程序的具体实现过程。

在 PC 机中，串口实际上可以有多种用途。一般情况下，串口用作一般的通信通路。在当 PC 机没有显示设备，比如 CRT 显示器的情况下，串口也可以用作系统控制台；在嵌入式的交叉调试环境中，串口也可以用作宿主机和目标机、调试器和调试代理之间的通信链路。

下面介绍的程序摘自某开源实时操作系统的源代码。它实现了串口操作的基本功能，包括端口寄存器的读写、数据的收发、中断驱动程序的安装等。

1．端口地址与操作数的宏定义

首先，是 16550 的各端口地址和操作数的宏定义，及该驱动程序对应的函数声明，这部分内容放在文件 uart.h 中（demo\chap6\uart.h），其详细定义如下：

```
/*
 * This software is Copyright (C) 1998 by T.sqware - all rights limited
 * It is provided in to the public domain "as is", can be freely modified
 * as far as this copyright notice is kept unchanged, but does not imply
 * an endorsement by T.sqware of the product in which it is included.
 */

#ifndef _BSPUART_H
#define _BSPUART_H
/*
        驱动程序的函数声明
*/
/*uart 芯片的初始化*/
void BSP_uart_init(int uart, int baud, int hwFlow);

/*设置传输的波特率*/
void BSP_uart_set_baud(int uart, int baud);

/*uart 芯片的中断控制*/
void BSP_uart_intr_ctrl(int uart, int cmd);

void BSP_uart_throttle(int uart);
void BSP_uart_unthrottle(int uart);
```

```
int   BSP_uart_polled_status(int uart);
void BSP_uart_polled_write(int uart, int val);
int   BSP_uart_polled_read(int uart);
void BSP_uart_termios_set(int uart, void *ttyp);
int   BSP_uart_termios_write_com1(int minor, const char *buf, int len);
int   BSP_uart_termios_write_com2(int minor, const char *buf, int len);

/*com1 口作为终端时的中断服务处理程序*/
void BSP_uart_termios_isr_com1();
/*com2 口作为终端时的中断服务处理程序*/
void BSP_uart_termios_isr_com2();

/*com1 口作为调试口时的中断服务处理程序*/
void BSP_uart_dbgisr_com1(void);
/*com2 口作为调试口时的中断服务处理程序*/
void BSP_uart_dbgisr_com2(void);

extern unsigned BSP_poll_char_via_serial(void);
extern void BSP_output_char_via_serial(int val);
extern int BSPConsolePort;
extern int BSPBaseBaud;
/* 为 BSP_uart_intr_ctrl()函数定义的命令参数 */
#define BSP_UART_INTR_CTRL_DISABLE    (0)
#define BSP_UART_INTR_CTRL_GDB        (0xaa)    /*RX only*/
#define BSP_UART_INTR_CTRL_ENABLE     (0xbb)    /*Normal operations*/
#define BSP_UART_INTR_CTRL_TERMIOS    (0xcc)    /*RX & line status*/

/*uart_polled_status()函数的返回值*/
#define BSP_UART_STATUS_ERROR    (-1)     /*No character*/
#define BSP_UART_STATUS_NOCHAR   (0)      /*No character*/
#define BSP_UART_STATUS_CHAR     (1)      /*Character present*/
#define BSP_UART_STATUS_BREAK    (2)      /*Break point is detected*/

/*PC 机串口和 UART 的定义*/
#define BSP_UART_COM1            (0)
#define BSP_UART_COM2            (1)

/*16550 对应串口 1、2 在 PC 机上的起始端口地址*/

#define COM1_BASE_IO 0x3F8
#define COM2_BASE_IO 0x2F8
```




```
/* 定义串口各个寄存器相对于起始端口地址的偏移量 */

/*DL 位为 0 时*/
#define RBR   (0)          /*Rx Buffer Register (read)*/
#define THR   (0)          /*Tx Buffer Register (write)*/
#define IER   (1)          /*Interrupt Enable Register */

/*DL 位为任意值时*/
#define IIR   (2)          /*Interrupt Ident Register (read)*/
#define FCR   (2)          /*FIFO Control Register (write)*/
#define LCR   (3)          /*Line Control Register*/
#define MCR   (4)          /*Modem Control Register*/
#define LSR   (5)          /*Line Status Register*/
#define MSR   (6)          /*Modem Status Register*/
#define SCR   (7)          /* Scratch register */

/*DL 位为 1 时*/
#define DLL   (0)          /*Divisor Latch, LSB*/
#define DLM   (1)          /*Divisor Latch, MSB*/
#define AFR   (2)          /* Alternate Function register*/

/*IIR 中断源的定义*/
#define MODEM_STATUS                          0
#define NO_MORE_INTR                          1
#define TRANSMITTER_HODING_REGISTER_EMPTY     2
#define RECEIVER_DATA_AVAIL                   4
#define RECEIVER_ERROR                        6
#define CHARACTER_TIMEOUT_INDICATION          12

/*IER 寄存器中“位”的定义*/
#define   RECEIVE_ENABLE              0x1
#define   TRANSMIT_ENABLE             0x2
#define RECEIVER_LINE_ST_ENABLE       0x4
#define MODEM_ENABLE                  0x8
#define INTERRUPT_DISABLE             0x0

/* ISR 寄存器中“位”的定义 */
#define DR         0x01     /*Data Ready*/
#define OE         0x02     /*Overrun Error*/
#define PE         0x04     /*Parity Error*/
#define FE         0x08     /*Framing Error*/
#define BI         0x10     /*Break Interrupt*/
```

```
#define THRE      0x20        /*Transmitter Holding Register Empty*/
#define TEMT      0x40        /*Transmitter Empty*/
#define ERFIFO    0x80        /*Error receive Fifo*/

/* MCR 寄存器中 "位" 的定义 */
#define DTR       0x01        /*Data Terminal Ready*/
#define RTS       0x02        /*Request To Send*/
#define OUT_1     0x04        /*Output 1, (reserved on COMPAQ I/O Board)*/
#define OUT_2     0x08        /*Output 2, Enable Asynchronous Port Interrupts*/
#define LB        0x10        /*Enable internal Loop Back*/

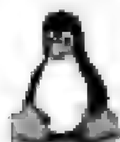
/* LCR 寄存器中 "位" 的定义 */
#define CHR_5_BITS 0
#define CHR_6_BITS 1
#define CHR_7_BITS 2
#define CHR_8_BITS 3

#define WL    0x03     /* Word length mask */
#define STB   0x04     /*1 Stop Bit, otherwise 2 Stop Bits*/
#define PEN   0x08     /*Parity Enabled*/
#define EPS   0x10     /*Even Parity Select, otherwise Odd*/
#define SP    0x20     /*Stick Parity*/
#define BCB   0x40     /*Break Control Bit*/
#define DLAB  0x80     /*Enable Divisor Latch Access*/

/* MSR 寄存器中 "位" 的定义 */
#define DCTS  0x01     /*Delta Clear To Send*/
#define DDSR  0x02     /*Delta Data Set Ready*/
#define TERI  0x04     /*Trailing Edge Ring Indicator*/
#define DDCD  0x08     /*Delta Carrier Detect Indicator*/
#define CTS   0x10     /*Clear To Send (when loop back is active)*/
#define DSR   0x20     /*Data Set Ready (when loop back is active)*/
#define RI    0x40     /*Ring Indicator (when loop back is active)*/
#define DCD   0x80     /*Data Carrier Detect (when loop back is active)*/

/* FIFO Control Register 寄存器中 "位" 的定义 ( WD16C552 or NS16550) */

#define FIFO_CTRL    0x01    /*Set to 1 permit access to other bits*/
#define FIFO_EN      0x01    /*Enable the FIFO*/
#define XMIT_RESET   0x02    /*Transmit FIFO Reset*/
#define RCV_RESET    0x04    /*Receive FIFO Reset*/
#define FCR3         0x08    /*do not understand manual!*/
```




```
#define RECEIVE_FIFO_TRIGGER1    0x0    /*trigger recieve interrupt after 1 byte*/
#define RECEIVE_FIFO_TRIGGER4    0x40   /*trigger recieve interrupt after 4 byte*/
#define RECEIVE_FIFO_TRIGGER8    0x80   /*trigger recieve interrupt after 8 byte*/
#define RECEIVE_FIFO_TRIGGER12   0xc0   /*trigger recieve interrupt after 12 byte*/
#define TRIG_LEVEL               0xc0   /*Mask for the trigger level*/

#endif /* _BSPUART_H */
```

2. 读写串口寄存器

有了上面的定义，就可以方便地对串口进行操作了。

接下来，介绍读写串口寄存器的两个函数 uread()和 uwrite()。其中，uread()函数用于从指定串口的指定寄存器读取一个字节，其参数 uart 指明了操作的串口，reg 指明了操作的寄存器。uwrite()函数用于向指定串口的指定寄存器写入一个字节。该函数多了一个 val 参数，表明要写入的值，其他的参数和 uread 一样。

uread()函数和 uwrite()函数的定义如下：

```
static inline unsigned char
uread(int uart, unsigned int reg)
{
  register unsigned char val;

  if (uart == 0) {/*根据参数决定是要操作哪个串口*/
    inport_byte(COM1_BASE_IO+reg, val);
  } else {
    inport_byte(COM2_BASE_IO+reg, val);
  }

  return val;
}

static inline void
uwrite(int uart, int reg, unsigned int val)
{
  if (uart == 0) {/*根据参数决定是要操作哪个串口*/
    outport_byte(COM1_BASE_IO+reg, val);
  } else {
    outport_byte(COM2_BASE_IO+reg, val);
  }
}
```

其中，outport_byte()是一个系统函数，其功能是向某个端口写入一个字节，该函数在i386平台上的定义如下：

```
#define outport_byte( _port, _value ) \
do { register unsigned short __port  = _port; \
     register unsigned char  __value = _value; \
     \
     asm volatile ( "outb %0,%1" : : "a" (__value), "d" (__port) ); \
   } while (0)
```

3. 串口的初始化

对串口的编程主要包括两部分：初始化和数据收发。串口的初始化编程主要完成对其工作模式的设定及内部寄存器的设置，包括设置波特率、数据帧格式和中断工作方式等。

BSP_uart_init()函数用于进行串口初始化，其入口参数包括：

- uart：表示要初始化的串口，可以选择 BSP_UART_COM1 或 BSP_UART_COM2。
- baud：欲设置的波特率，可以选择 50、75、110、134、300、600、1200、2400、9600、19200、38400、57600 和 115200 等。
- HwFlow：是否使用硬件流控，0 表示不使用，1 表示使用。

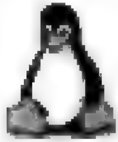
该函数中，串口数据帧格式被硬编码为 8 bit 数据位，没有奇偶校验位，1 bit 停止位，使用 FIFO 等。下面是它的代码：

```
int BSPBaseBaud     = 115200;
/*波特率最大值设置为 115200(16550 芯片的最高波特率)，它用于在后面 BSP_uart_init()
函数中计算波特率发生器除数寄存器的值*/


void
BSP_uart_init(int uart, int baud, int hwFlow)
{
  unsigned char tmp;

  /*检查入口参数 uart 的有效性*/
  assert(uart == BSP_UART_COM1 || uart == BSP_UART_COM2);

  /*检查入口参数 baud 的有效性*/
  switch(baud)
    {
    case 50:
    case 75:
    case 110:
    case 134:
```




```
    case 300:
    case 600:
    case 1200:
    case 2400:
    case 9600:
    case 19200:
    case 38400:
    case 57600:
    case 115200:
        break;
    default:
        assert(0);
        return;
    }

/*将 LCR 寄存器最高位置 1，以便下面访问波特率发生器除数寄存器*/
uwrite(uart, LCR, DLAB);

/*设置波特率低位，取波特率除数的低 8 位写入 DLL 寄存器*/
uwrite(uart, DLL,  (BSPBaseBaud/baud) & 0xff);
/*设置波特率高位，取波特率除数的高 8 位写入 DLM 寄存器*/
uwrite(uart, DLM,  ((BSPBaseBaud/baud) >> 8) & 0xff);

/*设置数据帧格式，8 bit 数据位、没有奇偶校验位、1 bit 停止位，如果需要设置其他
数据帧格式，只需修改此处即可*/
uwrite(uart, LCR, CHR_8_BITS);

/*设置调制解调器的标志位 DTR、RTS、OUT2 为 1，表示数据终端已准备好， 请求
发送，使能异步中断等*/
uwrite(uart, MCR, DTR | RTS | OUT_2);

/*使能 FIFO */
uwrite(uart, FCR, FIFO_EN | XMIT_RESET | RCV_RESET | RECEIVE_FIFO_TRIGGER12);

/*关闭中断*/
uwrite(uart, IER, 0);

/*通过读寄存器的方式，清除下面 3 个寄存器的值*/
tmp = uread(uart, LSR);
tmp = uread(uart, RBR);
tmp = uread(uart, MSR);
```

```
    /*保存状态值*/
    uart_data[uart].hwFlow      = hwFlow;
    uart_data[uart].baud        = baud;
    return;
}
```

4. 中断内的数据收发

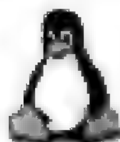
串口数据的收发，可以采用轮询方式，也可采用中断方式。使用中断方式可以提高CPU的利用率，因此在具有中断能力的系统中，一般采用中断方式。

在每次收发数据时，都要通过握手信号来判断通信链路的当前状态，以确保通信的可靠性。函数uart_termios_isr_com1()用于实现com1上的中断。该函数实际上是一个中断服务程序，每当16550控制芯片申请中断得到响应后，系统将转入该函数执行。它的安装过程将在后面描述。下面是它的代码：

```
void
uart_termios_isr_com1(void)
{
/**/
    unsigned char buf[40];
    unsigned char val;
    int        off, ret, vect;

    off = 0;
/*循环处理，直到所有中断被处理完为止*/
    for(;;)
      {
        /*读中断标识寄存器，确定中断源*/
      vect = uread(BSP_UART_COM1, IIR) & 0xf;

        /*根据中断源进行处理*/
      switch(vect)
        {
        /*调制解调器状态中中断处理方法，通过读调制解调器状态进行复位*/
case MODEM_STATUS :
/*读调制解调器状态*/
            val = uread(BSP_UART_COM1, MSR);
            /*如果启用了硬件流控功能，则根据CTS位进行相应处理*/
            if(uart_data[BSP_UART_COM1].hwFlow)
              {
                /*如果CTS为高，表示允许发送*/
                if(val & CTS)
```

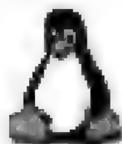



```
            {
                /* CTS high */
                termios_stopped_com1 = 0;
                if(termios_tx_hold_valid_com1)
                  {
                    termios_tx_hold_valid_com1 = 0;
                    BSP_uart_termios_write_com1(0, &termios_tx_hold_com1,
                                                1);
                  }
            }
/*如果 CTS 为低，表示停止发送*/
            else
            {
                /* CTS low */
                termios_stopped_com1 = 1;
            }
            }
          break;
    /*所有中断已经处理完*/
      case NO_MORE_INTR :
  /*如果串口输入的缓冲区的偏移不为 0，则将读入的字符加入到终端的输入队列中*/
        if(off != 0)
            {
              /* Update rx buffer */
              termios_enqueue_raw_characters(termios_ttyp_com1,
                                             (char *)buf,
                                             off);
            }
        /*所有中断已经处理完，返回*/
        return;
  /*发送缓存器为空，表示上一次的数据已经发送完成*/
      case TRANSMITTER_HODING_REGISTER_EMPTY :
        /*从中断得到要输出的字符*/
        ret = termios_dequeue_characters(termios_ttyp_com1, 1);

        /*如果没有字符需要输出，且硬件流控有效，则关闭 16550 的发送中断，使能接
收中断、线路状态中断和调制解调器中断*/
        if(ret == 0 && uart_data[BSP_UART_COM1].hwFlow)
          {
            /*关闭 16550 的发送中断*/
uwrite(BSP_UART_COM1, IER,
                (RECEIVE_ENABLE  |
```

```
                    RECEIVER_LINE_ST_ENABLE |
                    MODEM_ENABLE
                   )
                  );
                  termios_tx_active_com1 = 0;
              }
/*如果没有字符需要输出，且硬件流控功能关闭，则关闭 16550 的发送中断，使能接收
中断、线路状态中断*/
          else if(ret == 0)
            {
            /*关闭 16550 的发送中断*/
               uwrite(BSP_UART_COM1, IER,
                    (RECEIVE_ENABLE  |
                     RECEIVER_LINE_ST_ENABLE
                    )
                   );
                   termios_tx_active_com1 = 0;
            }
          break;
        /*接收器数据就绪，通过读数据进行复位*/
        case RECEIVER_DATA_AVAIL :
        /*字符超时，至少 4 个字节时间没有从接收器 FIFO 中读出数据，通过读数据进行
复位*/
        case CHARACTER_TIMEOUT_INDICATION:
          /*检查缓冲区空间是否已经溢出*/
          assert(off < sizeof(buf));
          /*读数据到缓冲区*/
          buf[off++] = uread(BSP_UART_COM1, RBR);
          break;
/*接收错误，通过读寄存器进行复位，实际上忽略了当前字符*/
        case RECEIVER_ERROR:
            uartError(BSP_UART_COM1);
          break;
        default:
          /*正常情况下，不会执行到此处*/
          assert(0);
          return;
        }
      }
}
```

函数 termios_isr_com1()中引用的 uartError()函数定义如下：




```
inline void uartError(int uart)
{
   unsigned char uartStatus;
   /*读取 LSR 和 RBR 的值，忽略当前接收错误的字符*/
   uartStatus = uread(uart, LSR);
   uartStatus = uread(uart, RBR);
}
```

5. 中断的安装与使能

在 PC 机上，16550 通过 8259 中断控制器向 CPU 请求中断。因此，为了使 16550 能够正常工作，必须安装对应的中断服务程序并正确设置 8259。图 6-4 说明了 16550 与 8259 的连接方式。

图 6-4　16550 与 8259 连接方法简图（串口 1）

从图 6-4 中可以看到，PC 机串口 1 连接到 8259 的中断引脚 4。因此，串口中断服务程序的中断类型号应该是 0×24=0×20+4，其中 0×20 是 8259 外部中断的起始值。

在保护模式下，安装中断服务程序，需要将中断服务程序的入口地址写入中断描述符表中，函数 void install_vector (unsigned vector,void* hdl,void** oldHdl)用于完成该功能。其中，参数 vector 代表中断类型号；hdl 代表中断服务程序的入口地址；oldHdl 代表原来的中断服务程序的入口地址。该函数代码如下：

```
/*中断描述符的定义*/
typedef struct {
   unsigned int low_offsets_bits : 16;
   unsigned int segment_selector : 16;
   unsigned int fixed_value_bits : 8;
   unsigned int gate_type    : 5;
   unsigned int privilege     : 2;
   unsigned int present              : 1;
   unsigned int high_offsets_bits: 16;
}interrupt_gate_descriptor;
```

```
void install_vector (unsigned vector,
                     void* hdl,
                     void** oldHdl)
{
    /*指向中断门描述符表的指针*/
    interrupt_gate_descriptor*  idt_entry_tbl;
    unsigned                    limit;
    /*临时的中断门描述符*/
    interrupt_gate_descriptor   new;
    unsigned int                level;

    /*得到中断门描述符表的信息，包括起始地址和大小*/
    i386_get_info_from_IDTR (&idt_entry_tbl, &limit);

    /*将 limit（中断门描述符表的大小）转化为中断门描述符的个数*/
    limit = (limit + 1) / sizeof(interrupt_gate_descriptor);

    /*检查中断类型号是否超出合法的中断门描述符的个数*/
    if (vector >= limit) {
      return;
    }
    /*关闭中断*/
    _CPU_ISR_Disable(level)

    /*保存旧的中断向量*/
    * ((unsigned int *) oldHdl) = idt_entry_tbl[vector].low_offsets_bits |
      (idt_entry_tbl[vector].high_offsets_bits << 16);
    /*填写新的中断门描述符*/
    create_interrupt_gate_descriptor(&new,  hdl);
    /*更新中断门描述符表，填入新的中断门描述符*/
    idt_entry_tbl[vector] = new;

    /*打开中断*/
    _CPU_ISR_Enable(level);
}
```

接下来，需要使能对应的中断，函数 int enable_at_i8259s(const int irqNum)用于使能 irqNum 对应的中断。该函数的代码如下：

```
int enable_at_i8259s(const int irqNum)
{
```




```
unsigned short mask;
unsigned int  level;

if ( ((int)irqLine < 0) || ((int)irqLine > 32) )
   return 1;
   /*关闭中断*/
 _CPU_ISR_Disable(level);

mask = ~(1 << irqLine);
i8259s_cache &= mask;

if (irqLine < 8)
{
   /*写主 8259 的中断屏蔽寄存器*/
   outport_byte(PIC_MASTER_IMR_IO_PORT, i8259s_cache & 0xff);
}
else
{
   /*写从 8259 的中断屏蔽寄存器*/
   outport_byte(PIC_SLAVE_IMR_IO_PORT, ((i8259s_cache & 0xff00) >> 8));
}
   /*打开中断*/
 _CPU_ISR_Enable (level);

 return 0;
}
```

到此为止，串口的驱动程序就介绍完了。串口的驱动程序实现的关键点在于：利用 8259 的一根中断请求线，通过在中断服务程序中查询的方式，实现了 4 种类型中断的处理，这也是中断管理中常用的方法。

6.4　小结

中断是现代计算机系统中微处理器与外部设备交换信息的一种方式，计算机在执行正常程序的过程中，会出现某些异常事件，例如外部设备请求 CPU 干预，或者执行了程序预先安排的软中断时，处理器会暂时中断正在执行的正常程序，而转去执行对应的中断服务程序。当处理完毕后，CPU 返回被暂时中断的程序，继续执行。

中断机制的引入，有效解决了 CPU 与外部设备之间在速度方面不匹配的问题。使用中断机制，可以提高 CPU 在处理外部设备，尤其是低速设备时的利用率。

6.5 思考题

1. 什么叫中断、中断源、中断请求和中断响应？
2. 什么是中断优先级和中断嵌套？
3. 什么叫中断向量表和中断描述符表？
4. 编写中断服务程序应该注意哪些问题？
5. 一般说来中断处理过程包括哪几个步骤？

第7章　嵌入式 Linux 下串口通信

知识点：

- 串口通信的基本概念
- RS-232 串口协议
- 嵌入式 Linux 下串口操作 API
- 扩展串行接口和串口操作例程

本章导读：

串口在嵌入式系统中有相当广泛的应用，本章将介绍嵌入式 Linux 下串口开发。首先，将对串口通信的一些基本概念加以介绍，如同步通信与异步通信，波特率、差错检测、DTE 与 DCE 通信等。然后，将介绍嵌入式 Linux 下实现对串口进行操作的 API 函数，以及此类 API 函数的使用方法。最后，将以在嵌入式系统中扩展一个串口为例，讲述串口开发方法，并给出一个串口通信例程。

7.1 串行 I/O 的基本概念

相信许多读者有过串口使用经验，如拨号上网的 Modem 就是通过串口和微机相连的。但是如果要在开发的嵌入式系统中利用串口进行通信，则必须对串口通信的数据格式、建立通信的过程、通信协议有清晰的了解。下面将对这些内容进行介绍。

7.1.1 同步通信与异步通信

在串行通信中，发送端或接收端每次都只发送或接收一位数据，即在某个时间，数据线上只有一位数据。串行通信可以分为同步通信和异步通信两种方式。不管是哪种通信方式都需要时钟信号，或者说计时参考，用于控制数据的流动。发送端需要一个时钟信号来决定什么时候发送一位数据到数据线上，接收端也需要时钟信号来决定什么时候从数据线上接收数据。它们的主要差别就是发送端和接收端的时钟是否是同步的，或是各自都有自己的时钟信号。下面分别对这两种通信方式加以介绍。

1. 同步通信

在同步通信中，发送端和接受端都使用同一个时钟，这个时钟可以由它们中的任意一个提供，也可以是外部时钟源。它的时钟图如图 7-1 所示。同步时钟可以有固定的频率，也可以每隔一个不规则的周期进行切换。所有要传输的位都与这个时钟信号同步，即每个传输的位在时钟跳变（上升沿或下降沿）之后的一个规定时间内有效。接收端也利用时钟跳变来决定何时读取一位数据。同步传输使用不同的方式来表示一次传输的开始和结束，包括起始位和停止位等。

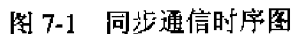


图 7-1　同步通信时序图

同步通信对于同一块电路板上各个部件之间或者使用很短的电缆连接是有用的，而且



速度也比异步传输要快。但是对于更长距离的连接，同步通信就不太适用了，因为需要传送时钟信号，这需要一根额外的时钟线，并且如果距离太长的话同步时钟很容易受到噪音的干扰。

2. 异步通信

在异步通信中，连接线不包括时钟线，时钟信号由发送端和接收端各自提供。因为连接的每一端都提供自己的时钟信号，所以每个终端的时钟频率必须保持一致，否则将产生失步。每个传输的字节都用一个起始位来与时钟同步，以及一个或几个停止位来表示传输字节的结束。串口通信中大多采用异步通信，如 PC 上的 RS-232 端口所使用的就是异步通信方式，PC 利用它和 Modem 或其他设备通信，虽然 RS-232 也支持同步通信方式，但异步通信更加普遍，大多数 RS-485 也是使用异步通信。

异步通信有多种格式，最通用的是 8-N-1，在这种方式中，发送方以一个起始位表示传输开始，后面紧跟 8 位数据，并以一个停止位表示一个字节传输结束。当接收方辨认出起始位后，就知道一个字节的传输开始了，并利用自己的时钟读取后面的 8 位数据，当接收到停止位后就停止读取，并把接收的数据送往接收缓冲。

8-N-1 中的 N 表示传输不使用奇偶校验位。在异步传输中也可以使用一位作为奇偶校验位，例如 7-E-1 就是这种格式。在发送端利用校验位保证发送的 7 位数据加上校验位必定有偶数个 1，接收端就可以利用这一特性判断传输过程是否出错。需要说明的是，传输数据的格式必须要由发送端和接收端共同约定好，接收方检验接收到的数据，如果不是期望的值，它就会向发送端发送出错通知。8-N-1 异步传输时钟图如图 7-2 所示。

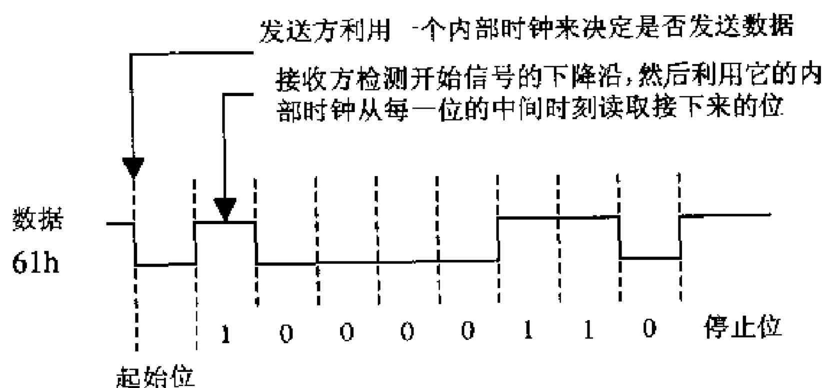


图 7-2　8-N-1 异步传输时序图

7.1.2　串口传输速率与流控

1. 串口速率

在串口传输中，常用波特率表示传输速度。波特（Baud）是码元传输速率的单位，一波特即为一秒钟传输一个码元。另一个表示传输速率的术语是比特率，它用于表示信息的

传输速率，单位是 bit/s，即一秒钟能传输多少信息。波特率与比特率在数量上有一定关系，若在编码方式上一个码元只携带一比特的信息量，则波特率等于比特率。但若一个码元携带有 n 比特的信息量，则 M Baud 的码元传输速率所对应的信息传输速率为 nM bit/s。例如，以 9600 的波特率传输的串口，若一个码元携带 2bit 的信息，则它的比特率为 19200bit/s。

所需要传输的数据包括起始位和停止位构成一个字，字中的数据位构成一个字符。在有些连接中，字符就是表示可以显示的正文字符（字母或者数字），而在其他连接中，字符是和正文无关的二进制数值。每秒传输的字符数等于波特率除以一个字的位数。由于在每个字中都包含有起始位和停止位，使得每个字节的传输时间增加了，例如在 8-N-1 方式中，每个字节的传输时间就增加了 25%（因为每个字节的传输有 10 位而不是 8 位），所以，1 字节的传输速率等于波特率的 1/10，一个 9600 bit/s 的连接每秒传输 960 个字节。

2. 流控

这里讲到的“流”，指的是数据流。数据在两个串口之间传输时，常常会出现丢失数据的现象，或者两台计算机的处理速度不同，如台式机与单片机之间的通信，接收端数据缓冲区已满，则此时继续发送来的数据就会丢失。例如，在网络上通过 Modem 进行数据传输，这个问题就尤为突出。流控制可以解决这个问题，当接收端数据处理不过来时，就发出“不再接收”的信号，发送端就停止发送，直到收到“可以继续发送”的信号再发送数据。因此流控制可以控制数据传输的进程，防止数据的丢失。PC 机中常用的两种流控制分别是硬件流控制（包括 RTS/CTS、DTR/CTS 等）和软件流控制 XON/XOFF（继续/停止）。下面将分别介绍不同的流控制方式。

（1）硬件流控。

硬件流控发生在使用多线电缆直接相连的系统之间。使用 1~2 根线进行数据传送，其他的线用于信令传送。例如，在使用请求发送/允许发送（RTS/CTS）信令方法中，一台终端设备通过使其 RTS 线有效表明可以发送数据，其他设备使其 CTS 线有效作为响应，为实现流控，接收者可以随时关闭其 CTS 线，当它准备好了接收数据时，就把这根线的电平置高，若没有准备好，就置低电平。

这种硬件握手方式的过程为：在编程时根据接收端缓冲区大小设置一个高位标志（可为缓冲区大小的 75%）和一个低位标志（可为缓冲区大小的 25%），当缓冲区内数据量达到高位时，在接收端将 CTS 线置低电平（送逻辑 0），当发送端的程序检测到 CTS 为低后，就停止发送数据，直到接收端缓冲区的数据量低于低位而将 CTS 置高电平。RTS 则用于标明发送设备请求发送数据。

常用的流控制信号还有 DTR/DSR（数据终端就绪/数据设置就绪），它的原理与 CTS/RTS 差不多，对此将不再详述。

（2）软件流控。

由于电缆线的限制，普通的控制通信中一般不用硬件流控制，而用软件流控制。一般通过 XON/XOFF 来实现软件流控制。常用方法是：当接收端的输入缓冲区内数据量超过设定的高位时，就向数据发送端发出 XOFF 字符（十进制的 19 或 Control-S），发送端收到 XOFF 字符后就立即停止发送数据；当接收端的输入缓冲区内数据量低于设定的低位时，



就向数据发送端发出 XON 字符（十进制的 17 或 Control-Q），发送端收到 XON 字符后就立即开始发送数据。

应该注意，若传输的是二进制数据，标志字符也有可能在数据流中出现而引起误操作，这是软件流控制的缺陷，而硬件流控制不会有这个问题，但是 CTS/RTS 并不支持所有的硬件和操作系统。

由于流控制的多样性，当系统里用了流控制时，应作详细的说明，包括如何接线、如何应用等。

7.1.3 差错控制

当一台计算机想要发送数据时，接收方可能正被其他的任务占据着，无法立刻接收数据，在这种情况下必须采取相应的措施以防止数据丢失，如前面介绍的流控就是其中的一种措施。其他的措施还包括缓存、轮询或中断、错误检测等。下面将对这些差错控制方法进行说明。

1. 缓存

在串行通信中，缓存对于发送方和接收方都有重要意义。在发送方使用缓存，可以把要发送的数据先存储起来，等连接可用时集中发送，这样可以使应用程序更有效地工作。对于接收方，当应用程序来不及接收数据时，可以使用缓存把数据暂时缓存起来，这样可以防止数据丢失。

缓存可以是硬件的，也可以是软件的，或者两者都使用。在老式的 PC 串口上都有 16 个字节的硬件缓存内嵌在 UART 中。这在接收方意味着 UART 可以在软件读取它之前存储最多 16 个字节的数据。在发送的方向上，UART 可以存储最多 16 个字节的数据，然后 UART 将会负责对这些数据的按位发送。

当硬件缓存不够大时，也可以使用软件缓冲，它的大小是可以编程的。端口的软件缓冲用于在硬件和软件驱动之间传送数据。

2. 轮询和中断

在一个串口端口发生的事件包括发送和接收数据、握手信号的改变以及错误通知等，应用程序如何知道这些事件的发生呢？或者说如何通知应用程序这些事件已经发生了，以防止数据丢失呢？可以通过中断和轮询的方式。

在第 6 章已经对中断作了相应介绍，它的处理方法就是当一个事件发生时，自动跳到处理程序中执行。应用程序对端口的行为反应非常迅速而且是自动的，不需要浪费时间进行检测。这种类型的编程称为事件驱动，因为一个外部的事件可以在任何时间插入并且使得程序的执行转向另一个代码分支。

另一种方法就是轮询端口，它通过周期性的读取特征信号来发现事件是否发生。这种类型的编程称为过程编程，并且不使用端口的硬件中断。这种编程方式需要确保对端口的轮询足够频繁，以保证不会遗失任何事件和数据。轮询的频率取决于缓存的大小和希望的

数据量（以及对快速反应的需求）。例如，如果一个设备有 16 个字节的缓存并且每秒钟轮询 2 次端口缓存，则它每秒接收不能超过 32 个字节的数据，不然缓冲区就会溢出，从而导致数据丢失。

轮询通常适合于传输短的字符组，或者在计算机发送数据后希望立即得到答复的情况下。由于轮询不需要硬件中断，因此可以在一个没有分配硬件中断的端口使用轮询。很多轮询接口都使用系统计时中断来确定周期性读取端口操作的时间。

3. 差错检测

一个接收者可以使用差错检测来检验所有数据是否正确到达，对一条消息进行差错检验的方法包括发送冗余数据和差错检验字节。

差错检验的一个最简单的形式是使用冗余或者副本数据，发送者对每条数据发送两次，然后接收者对接收到的两次数据进行比较，看是否一致。当然，这种方法意味着每条消息要花两倍时间进行传输。它在传输小数据量短字符组时仍然有用，许多红外线控制器都使用这种方法。

另一种差错检验的方法是和数据一道发送一个差错检验字节。通过对一条信息中的字节进行数学或者逻辑操作可以计算出校验和。一个典型的计算是将这条消息中所有的字节的值加起来，然后用最低的一个字节作为校验和。接收端重复这样的计算，如果得到一个不同的结果，就知道传输出错。

差错检验字节的另一种形式是使用 CRC（循环冗余编码），它使用更为复杂的数学变换并且比校验和更加可靠。

当接收端检测到一个差错或者接收到一条它无法理解的消息时，它应当通知发送端使它能够重新发送或者采取别的行动来校正这种情况。经过数次发送，仍然出错或者接收端没有回复，发送端就应该发送一条出错消息，报警或者进行其他操作来通知这个问题的操作人员进行人工干预，然后尽可能地不影响继续其他任务。接收端也应该知道如果一条消息比预期的短该如何处理，它应当终止连接，然后通知主机出现了问题，而不是无休止地等待一个消息。

7.1.4 DTE 和 DCE 通信过程

在数据通信领域，根据通信的位置不同有 DTE 和 DCE 之分。下面将对 DTE 与 DCE 通信进行相应的介绍。

DTE（Data Terminal Equipment）是数据终端设备，它是具有一定数据处理能力以及发送和接收数据能力的设备。大家都知道，大多数数据处理设备的数据传输能力都是有限的。直接将两个数据处理设备在很远的距离连接起来是不能通信的，必须在 DTE 和传输线路之间加上一个中间设备，这个设备就是数据电路端接设备 DCE（Data Circuit-terminating Equipment），DCE 的作用就是在 DTE 和传输线路之间提供信号变换和编码的功能，并且负责建立、保持和释放数据链路的连接。如图 7-3 所示为 DTE 通过 DCE 连接到通信传输线



路的情况。

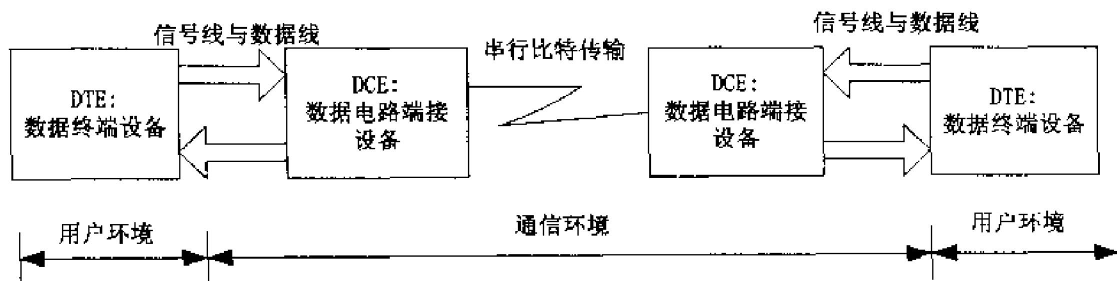


图 7-3 DTE 通过 DCE 与通信传输线路相连

DTE 可以是一台计算机，也可以是各种 I/O 设备。典型的 DCE 则是一个与模拟电话线路相连的调制解调器（MODEM）。DTE 和 DCE 之间一般有许多根线，包括数据线和控制线。DCE 将 DTE 传过来的数据按比特顺序逐个发往传输线路，或者反过来，从传输线路接收下来串行的比特流然后交给 DTE。下面以使用 Modem 进行通信为例对串口的 DTE 与 DCE 通信过程加以说明，它们的连接图如图 7-4 所示。

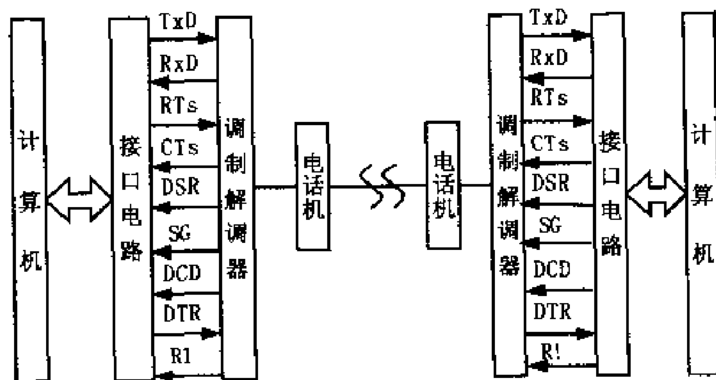


图 7-4 DTE 与 DCE 通信连接图

注意：这里所说的发送和接收都是相对于 DTE 而言的。若在双方 Modem 之间采用普通电话交换线进行通信，除了需要 2～8 号信号线外还要增加 RI（22 号）和 DTR（20 号）两个信号线进行联络。

若 DTE 和 DCE 已经准备好，则 DTR 和 DSR 信号分别有效，它们表示设备本身已经可用，接下来就可以开始建立通信了，但这时通信并没有建立。

首先，通过电话机拔号呼叫对方，电话交换台向对方发出拔号呼叫信号，当对方 DCE 收到该信号后，使 RI（振铃信号）有效，通知 DTE 已被呼叫。当对方“摘机”后，两方就建立了通信链路。

若计算机要发送数据至对方，则首先通过接口电路（串口）发出 RTS（请求发送）信号。此时，若 DCE（在这里是 MODEM）允许传送，则向 DTE 回答 CTS（允许发送）信号。一般可直接将 RTS/CTS 接高电平，即只要通信链路已建立，就可传送数据（RTS/CTS

可只用于半双工系统中作发送方式和接收方式的切换)。

当 DTE 获得 CTS 信号后，就可以通过 TXD 线向 DCE 发出串行数据了，DCE(MODEM)将这些数字信号调制成模拟信号（又称载波信号），传给对方。

应用程序在向 DTE 的数据发送寄存器传送新的数据前，应检查 DCE（MODEM）状态是否正常和数据输出寄存器是否为空。当对端的 DCE 收到载波信号后，就向对端的 DTE 发出 DCD 信号（数据载波检出），通知其 DTE 准备接收，同时，将载波信号解调为数据信号，从 RXD 线上送给 DTE，DTE 通过串行接收移位寄存器对接收到的位流进行移位，当收到 1 个字符的全部位流后，把该字符的数据位送到数据输入寄存器，CPU 可以从数据输入寄存器读取字符。于是远程 DTE 之间的通信就建立起来了。

7.1.5 RS-232C 串口规范简介

RS-232C 标准是美国 EIA（电子工业联合会）与 BELL 等公司一起开发的通信协议。它适合于数据传输速率在 0～20000bit/s 范围内的通信。这个标准对串行通信接口的有关问题，如信号线功能、电气特性都作了明确规定。由于通行设备厂商都生产与 RS-232C 制式兼容的通信设备，因此，它作为一种标准，目前已在微机通信接口中得到了广泛采用。

串行通信接口标准经过使用和发展，目前已经有几种，但都是在 RS-232C 标准的基础上经过改进而形成的。RS-232C 以其编程方便、价格便宜等优点，在嵌入式系统中有广泛的应用。所以，本书将以 RS-232C 为主来讨论。下面分别对它的电气特性、信号线功能加以介绍。

1. 电气特性

对于 RS-232C 规范的电气特性，它们表示数据 1 和数据 0 或者信号有效和无效的电平特性分别为：

在 TxD 和 RxD 上：

逻辑 1（MARK）=-3V～-15V；逻辑 0（SPACE）=+3～+15V

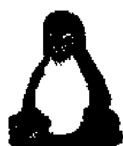
在 RTS、CTS、DSR、DTR 和 DCD 等控制线上：

信号有效（接通，ON 状态，正电压）=+3V～+15V；

信号无效（断开，OFF 状态，负电压）=-3V～-15V。

从以上的规定中可以看出，对于数据（信息码）：逻辑“1”的电平应该低于-3V，逻辑“0”的电平高于+3V。对于控制信号：接通状态（ON）即信号有效的电平应该高于+3V，断开状态（OFF）即信号无效的电平应该低于-3V。所以当传输电平的绝对值大于 3V 时，电路可以有效地检查出来，介于-3～+3V 之间的电压则无意义，而低于-15V 或高于+15V 的电压也认为无意义，因此，在实际工作时应保证电平在±(3～15)V 之间。

由于 EIA-RS-232C 是用正负电压来表示逻辑状态的，与 TTL 以高低电平表示逻辑状态的规定不同。因此，为了能够同计算机接口或终端的 TTL 器件连接，必须在 EIA-RS-232C 与 TTL 电路之间进行电平和逻辑关系的变换。



实现这种变换的方法可用分立元件，也可用集成电路芯片。目前较为广泛地使用集成电路转换器件，如 MC1488、SN75150 芯片，它们可完成 TTL 电平到 EIA 电平的转换，而 MC1489、SN75154 可实现 EIA 电平到 TTL 电平的转换。MAX232 芯片可完成 TTL←→EIA 双向电平转换。

例如，若使用 MC1488 及 MC1489 进行转换，MC1488 的引脚 2、4、5、9、10、12、13 接输入；MC1488 的引脚 3、6、8、11 接输出；MC1498 的 14 的 1、4、10、13 脚接 EIA 输入，而 3、6、8、11 脚接 TTL 输出。具体连接方法如图 7-5 所示。图中的左边是微机串行接口电路中的主芯片 UART，它是 TTL 器件，右边是 EIA-RS-232C 连接器，要求 EIA 高电压。因此，RS-232C 所有的输出、输入信号都要分别经过 MC1488 和 MC1498 转换器进行电平转换后才能送到连接器上去或从连接器上送进来。

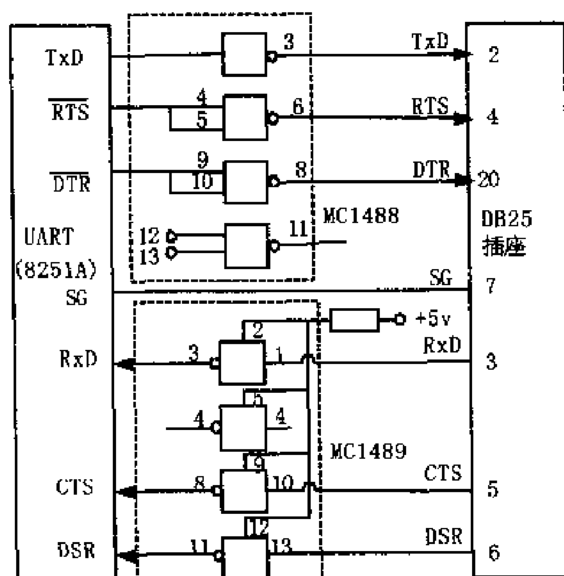


图 7-5　EIA-RS-232C 与 TTL 转换连接图

由于 RS-232C 并未定义连接器的物理特性，因此，出现了 DB-25、DB-15 和 DB-9 各种类型的连接器，其引脚的定义也各不相同。下面将分别重点介绍 DB-25 和 DB-9 这两种连接器。

（1）DB-25。

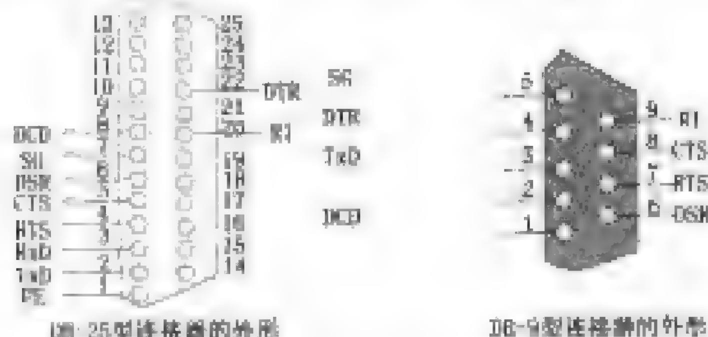
PC 和 XT 机都采用 DB-25 型连接器。DB-25 连接器定义了 25 根信号线，分为 4 组：

- 异步通信的 9 个电压信号（含信号地 SG）：2、3、4、5、6、7、8、20、22。
- 20mA 电流环信号 9 个：12、13、14、15、16、17、19、23、24。
- 空 6 个：9、10、11、18、21、25。
- 保护地（PE）1 个，作为设备接地端：1。

注意：DB-25 接口使用的 20mA 电流环信号仅早期的 IBM PC 和 IBM PC/XT 机提供，至 AT 机及其以后系列，已不支持。

（2）DB-9。

在 AT 机及其以后，已不支持 20mA 电流环接口，而使用 DB-9 连接器，作为提供多功能 I/O 卡或主板上 COM1 和 COM2 两个串行接口的连接器，它只提供异步通信的 9 个信号。DB-9 型连接器的引脚分配与 DB-25 型引脚信号有着巨大差别。因此，它若需要与配接 DB-25 型连接器的 DCE 设备连接，必须使用专门的电缆线。它们的外形如图 7-6 所示。



DB-25型连接器的外形　　DB-9型连接器的外形

图 7-6　DB-25 型与 DB-9 连接器的外形

RS-232 协议的传输距离为：

- 电缆长度：在通信速率低于 20kbit/s 时，RS-232C 所直接连接的最大物理距离为 15m（50 英尺）。
- 最大直接传输距离：RS-232C 标准规定，若不使用 Modem，在码元畸变小于 4%的情况下，DTE 和 DCE 之间最大传输距离为 15m（50 英尺）。可见这个最大的距离是在码元畸变小于 4%的前提下给出的。为了保证码元畸变小于 4%的要求，接口标准在电气特性中规定，驱动器的负载电容应小于 2500pF。

2. 信号线

RS-232C 规定标准接口有 25 条线，其中 4 条数据线、11 条控制线、3 条定时线、7 条备用和未定义线，但常用的只有 9 根。每条信号线的功能如表 7-1 所示。

表 7-1　RS-232 信号线功能说明

232 引脚	CCITT	MODEM	名　称	说　明	用途	
					异　步	同　步
1	101	AA	保护地	设备外壳接地	PE	PE√
2	103	BA	发送数据	数据送 Modem	TXD	
3	104	BB	接收数据	从 Modem 接收数据	RXD	
4	105	CA	请求发送	在半双工时控制发送器的开和关	RTS	
5	106	CB	允许发送	Modem 允许发送	CTS	
6	107	CC	数据终端准备好	Modem 准备好	DSR	
7	102	AB	信号地	信号公共地	SG	SG√



（续）

232 引脚	CCITT	MODEM	名　称	说　明	用　途	
					异　步	同　步
8	109	CF	载波信号检测	Modem 正在接收另一端送来的信号	DCD	
9			空			
10			空			
11			空			
12			接收信号检测（2）	在第二通道检测到信号		√
13			允许发送（2）	第二通道允许发送		√
14	118		发送数据（2）	第二通道发送数据		√
15	113	DA	发送器定时	为 Modem 提供发送器定时信号		√
16	119		接收数据（2）	第二通道接收数据		√
17	115	DD	接收器定时	为接口和终端提供定时		√
18			空			
19			请求发送（2）	连接第二通道的发送器		√
20	108	CD	数据终端准备好	数据终端准备好	DTR	
21			空			
22	125		振铃	振铃指示	RI	
23	111	CH	数据率选择	选择两个同步数据率		√
24	114	DB	发送器定时	为接口和终端提供定时		√
25			空			

注：表中打“√”表示同步传输用到的信号线，这里假设 DCE 为 Modem。

7.2 编写串口通信程序

在嵌入式 Linux 应用系统中，如果要利用串口作为通信接口，则需要编写串口通信程序，在本节中将介绍编写串口通信程序的相关知识。

7.2.1 嵌入式 Linux 驱动程序简介

在本章将开始介绍在嵌入式 Linux 下如何编写设备驱动程序。设备驱动程序是操作系统内核和机器硬件之间的接口。设备驱动程序为应用程序屏蔽了硬件的细节，这样使硬件对应用程序来说是透明的，在应用程序看来，硬件设备只是一个设备文件，应用程序可以像操作普通文件一样对硬件设备进行操作。设备驱动程序是嵌入式 Linux 内核的一部分，它完成以下的功能：

- 对硬件设备初始化和释放。
- 把数据从内核传送到硬件，或从硬件读取数据。
- 读取应用程序传送给设备文件的数据和回送应用程序请求的数据。
- 检测和处理设备出现的错误和异常。

Linux 系统的设备分为字符设备（char device）、块设备（block device）和网络设备（network device）3 种。字符设备是指存取时没有缓存的设备，它不使用系统缓冲，对设备文件的操作直接反映到硬件上。块设备的读写都有缓存来支持，并且块设备必须能够随机存取（random access），字符设备则没有这个要求。典型的字符设备包括鼠标、键盘、串行口等。块设备主要包括硬盘软盘设备、CD-ROM 等。网络设备在嵌入式 Linux 里作专门的处理。Linux 的网络系统主要是基于 BSD UNIX 的 Socket 机制。

用户进程通过设备文件实现与硬件的交流。每个设备文件都有其文件属性，表示是字符设备还是块设备。另外每个文件都有两个设备号：第一个是主设备号，标识驱动程序；第二个是从设备号，标识使用同一个设备驱动程序的不同的硬件设备，比如有两个软盘，就可以用从设备号来区分它们。设备文件的主设备号必须与设备驱动程序在登记时申请的主设备号一致，否则用户进程将无法访问到驱动程序。

设备驱动程序可以分为 3 个主要组成部分：

（1）自动配置和初始化子程序。

用于负责检测所要驱动的硬件设备是否存在和是否能正常工作。如果该设备正常，则对这个设备及其相关的设备驱动程序需要的软件状态进行初始化，如设置寄存器的值，初始化驱动程序用到的数据结构。这部分驱动程序仅在初始化的时候被调用一次。

（2）服务于 I/O 请求的子程序。

调用服务于 I/O 请求的子程序是由于系统调用的结果，如 read、write 调用。这部分程序在执行的时候，系统仍认为是和进行调用的进程属于同一个进程，只是由用户态变成了核心态，它们的运行环境和进行此系统调用的用户程序一样，因此可以在其中调用 sleep() 等与进程运行环境有关的函数。

（3）中断服务子程序。

在 UNIX 系统中，并不是直接从中断向量表中调用设备驱动程序的中断服务子程序，而是由 UNIX 系统来接收硬件中断，再由系统调用中断服务子程序。中断可以发生在任何一个进程运行的时候，因此在中断服务程序被调用的时候，不能依赖于任何进程的状态，也就不能调用任何与进程运行环境有关的函数。因为设备驱动程序一般支持同一类型的若干设备，所以一般在系统调用中断服务子程序的时候，都带有一个或多个参数，以惟一标识请求服务的设备。

在系统内部，I/O 设备的存取通过一组固定的入口点来进行，这组入口点是由每个设备的设备驱动程序提供的。一般来说，字符型设备驱动程序能够提供如下几个入口点：

- open 入口点

open 入口点用于打开设备准备 I/O 操作。对字符设备文件进行打开操作，都会调用设备的 open 入口点。open 子程序必须对将要进行的 I/O 操作做好必要的准备工作，如清除缓冲区等。如果设备是独占的，即同一时刻只能有一个程序访问此设备，则 open 子程序必须



设置一些标志以表示设备处于忙状态。

● close 入口点

close 入口点用于关闭一个设备。当最后一次使用设备终结后，调用 close 子程序。独占设备必须标记设备可再次使用。

● read 入口点

read 入口点用于从设备上读数据。对于有缓冲区的 I/O 操作，一般是从缓冲区里读数据。对字符设备文件进行读操作时，将调用 read 子程序。

● write 入口点

write 入口点用于往设备上写数据。对于有缓冲区的 I/O 操作，一般是把数据写入缓冲区里。对字符设备文件进行写操作将调用 write 子程序。

● ioctl 入口点

ioctl 入口点用于执行读、写之外的操作。

● select 入口点

select 入口点用于检查设备，查看数据是否可读或设备是否可用于写数据。select 系统调用在检查与设备文件相关的文件描述符时使用 select 入口点。

如果设备驱动程序没有提供上述入口点中的某一个，系统会用默认的子程序来代替。对于不同的系统，还有一些其他的入口点。

下面通过一个实例说明嵌入式 Linux 下驱动程序的编写。该实例将实现在加载时打印“Hello,I'm in kernel mode”提示信息，退出时打印“Hello，I'm going to out”提示信息。当调用 read()读取数据时，把缓冲区全部置为 1。同时，在该实例中，只实现 open()、close()和 read()入口点（见光盘/demo/chap07/7-1）。

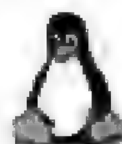
```
/*TestDriver.c*/
/*一个简单的字符设备驱动实例*/
#define __NO_VERSION__
#include <linux/modules.h>
#include <linux/version.h>
char kernel_version [] = UTS_RELEASE;
#define KERNEL
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/errno.h>
#include <asm/segment.h>
#define SUCCESS 0
static int device _read(struct inode *, struct file *, char *, int);
static int device_open(struct inode *inode, struct file *file)
static void device _release(struct inode *, struct file *);
/*在 fs.h 中定义与入口相关的数据结构*/
```

```
struct file_operations tdd_fops =
{
        NULL,
         device _read,
          NULL,
          NULL,
          NULL,
          NULL,
          NULL,
         device _open,
         device _release,
          NULL,
          NULL,
          NULL,
          NULL
};
/*设备声明 ********************************* */
/*将出现在 /proc/devices 中设备名*/
#define DEVICE_NAME "char_dev"
/*设备正打开? 防止对同一设备的同时访问*/
static int Device_Open = 0;
unsigned int test_major = 0;
/*当被询问时设备将给出的消息*/
static char Message[BUF_LEN];
/*这个函数在进程试图打开设备文件时被调用*/
static int device_open(struct inode *inode, struct file *file){
            #ifdef DEBUG
            printk ("device_open(%p)\n", file);
            #endif

            /*不想同时和两个进程对话*/
            if (Device_Open)
            return -EBUSY;

            Device_Open++;
            MOD_INC_USE_COUNT;
            return SUCCESS;
}

    /*这个函数在关闭设备时调用*/
    static void device_release(struct inode *inode, struct file *file){
            #ifdef DEBUG
```




```
                printk ("device_release(%p,%p)\n", inode, file);
                #endif

    /*为下个调用者做准备*/
                Device_Open --;
                MOD_DEC_USE_COUNT;
}
    /*read()入口，这个函数是为 read 调用准备的，当调用 read 时，read_test()被调用，它把用户的缓冲区全部写 1*/
static int device_read (struct inode *node,struct file *file,char *buf,int count){
                int left;
                if (verify_area(VERIFY_WRITE,buf,count) == -EFAULT )
                return -EFAULT;
                for(left = count ; left > 0 ; left--){
                /*在 device_read 被调用时，系统进入核心态，所以不能使用 buf 这个地址，而必须用_put_user()，这是 kernel 提供的一个函数，用于向用户传送数据。另外还有很多类似功能的函数，请参考<linux/mm.h>*/
                _put_user(1,buf,1);
                buf++;
    }
                return count;
}
```

注意：必须把要实现的入口函数在 file_operations 结构中加以声明。

到此，设备驱动程序的主体部分可以说是写好了。现在需要把驱动程序嵌入 Linux 内核。驱动程序可以按照两种方式编译：一种是直接编译进内核（kernel），另一种是编译成模块（module）。如果编译进内核的话，不仅会增加内核的容量，还需要改动内核的源文件，而且不能动态地卸载，不利于调试。所以，在通用的 Linux 中一般使用模块加载的方式。但是在嵌入式 Linux 中，由于一般不支持动态模块加载的方式，所以只能使用编译进内核的方式来加载驱动。在这里为了说明一般意义上的驱动编写方法，还是采用动态模块加载。加载的代码如下：

```
/*init_module 向系统的字符设备表登记一个字符设备*/
int init_module(void)
{
            int result;
            /*注册字符设备*/
            result = register_chrdev(0, " char_dev", & sdd_fops);
            if (result < 0) {
            printk(KERN_INFO "char_dev: can't get major number\n");
            return result;
```

```
    }
/*如果登记成功，则返回0，得到设备的主设备号；如不成功，则返回一个负值*/
if (test_major == 0) test_major = result;
/*安装进入内核时打印提示信息*/
printk("Hello,I'm in kenel mode\n");
 return 0;
}
```

使用 insmod 命令将编译好的模块调入内存，用 rmmod 命令卸载模块。在用 insmod 命令将编译好的模块调入内存时，init_module()函数将被调用。在这里，register_chrdev()需要3个参数，第1个参数表示希望获得的设备号，如果是0，系统将选择一个没有被占用的设备号返回；第2个参数表示设备文件名；第3个参数用于登记驱动程序实际执行操作的函数的指针。

```
void cleanup_module(void)
{
printk("Hello，I'm  going to out \n");
unregister_chrdev(test_major, "test");
}
```

在用 rmmod 命令卸载模块时，cleanup_module()函数将被调用，它用于释放字符设备 test 在系统字符设备表中所占有的表项。

至此，一个简单的字符设备的驱动程序已编写完成。下面执行以下命令对其加以编译：

$ gcc -O2 -DMODULE -D__KERNEL__ -c TestDriver.c

经过编译得到的文件 TestDriver.o 就是一个设备驱动程序。

再执行以下命令把它安装到系统中：

$ insmod -f TestDriver.o

如果安装成功，在/proc/devices 文件中就可以看到设备 char_dev，并可以看到它的主设备号。

如果需要卸载，可运行以下命令：

$ rmmod char_dev

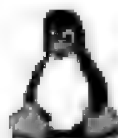
下面就需要创建设备文件了。可执行以下命令：

mknod /dev/ char_dev c major minor

其中，选项 c 是指字符设备；major 是主设备号，可以在/proc/devices 里看到；minor 是从设备号，将其设置成0即可。

现在就可以通过设备文件来访问驱动程序了。可以通过以下程序加以测试。

```
#include <stdio.h>
#include <sys/types.h>
```




```
#include <sys/stat.h>
#include <fcntl.h>
main()
{
        int testdev;
        int i;
        char buf[10];
        testdev = open("/dev/char_dev",O_RDWR);
        if ( testdev == -1 )
        {
        printf("Cann't open file \n");
        exit(0);
        }
        read(testdev,buf,10);
        for (i = 0; i < 10;i++)
        printf("%d\n",buf[i]);
        close(testdev);
}
```

编译运行后，看看是不是打印出全 1。

到此，整个字符驱动程序的编写就已完成，这可能会给读者一个假象，即驱动程序的编写是一件非常简单的工作。事实恰恰相反，通常驱动程序的编写是一个应用系统的难点，它牵涉到软硬件两方面，要求程序员对软硬件都非常熟悉。而且进入操作系统内核时，稍不注意就会导致死机，调试也不十分方便。在本章中只是实现了驱动程序最简单的部分，目的是让读者了解嵌入式 Linux 下驱动程序的基本结构。至于驱动程序中的中断、定时器、内存分配和代码重入等难点都没有碰到，有关这方面的内容将在本书的其他章节加以讨论。

7.2.2 串口访问函数

在 7.2.1 节中介绍了在嵌入式 Linux 下编写设备驱动程序的方法，事实上在大多数嵌入式 Linux 操作系统中，都已经带有常用设备的驱动程序，如一般的操作系统都带有串口的驱动程序，只需调用系统提供的操作函数就可以访问相应的串口，除非对串口有特殊要求，否则不必另外开发串口驱动程序。

同所有的设备一样，嵌入式 Linux 是通过设备文件访问串口的。在访问具体的串行端口时，只需打开相应的设备文件即可，在嵌入式 Linux 系统中，串行端口 Port1 和 Port2 对应的设备文件分别为：/dev/ttyS0 和/dev/ttyS1。下面将对串口访问函数进行相应的介绍。

1. 打开串口

类似于普通文件，串口也可以使用它的设备文件，用 open()函数打开。当然，执行打

开操作的用户必须有相应的权限。

下面给出一段代码，说明如何在 Linux 平台下执行串口打开操作。

```
#include <stdio.h>          /*标准输入输出定义*/
#include <string.h>         /*字符串功能定义*/
#include <unistd.h>         /*UNIX 标准函数定义*/
#include <fcntl.h>          /*文件控制定义*/
#include <errno.h>          /*错误号定义*/
#include <termios.h>        /*POSIX 终端控制定义*/

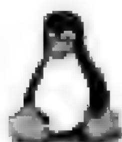
/*
open_port() — 打开串行口 1
成功则返回文件描述符，否则返回-1
*/

int open_port(void)
{
        int fd;   /*端口文件描述符*/

        fd=open("dev/ttyS0",O_RDWR|O_NOCTTY|O_NDELAY);
        if(fd==-1)
        {
              /*打开串口出错，无法打开*/
              perror("open_port:Unabel to open /dev/ttyS0");
        }
        else
        fcntl(fd,F_SETFL,0);
 }
        return(fd);
 }
```

使用 open()函数执行打开操作，open()函数带有 2 个参数，其中参数 1 为要打开的设备文件名，参数 2 为打开方式，它们的含义为：

- O_RDWR：可读可写。
- O_NOCTTY：告诉嵌入式 Linux，该程序不想成为此端口的"控制终端"。如果应用程序不强调这一点，那么任何输入（例如键盘的中断信号等）都会影响程序的执行。
- O_NDELAY：表示该程序不关注 DCD 信号线所处的状态，即不管对端设备是在运行或是挂起。如果不设置该标志，则程序会被设置为睡眠状态，直到 DCD 信号为低为止。



2. 关闭串口

使用 close()系统调用关闭串口。调用方式为：

```
close(fd); /*fd 为打开时得到的文件描述符*/
```

关闭一个串口通常会将 DTR 信号设置为低电平，如果外接 Modem，就会将 Modem 挂起。

3. 向串口写数据

使用 write()函数向串口写入数据，如要将存储在 W_buf 缓冲区内的“How are you”通过串口发送，使用如下方式：

```
int n;
n=write(fd,W_buf,11);
if(n!=11)
fputs("write operation error!\n");
```

write 调用若正确，返回发送的字节数，否则返回-1。

4. 从串口读取数据

从串口读取数据需要一定技巧。如果在原始数据的模式下对端口进行操作，read()函数调用将返回串行口输入缓冲区所有的数据，而不管多少。如果没有数据，那么该进程将被阻塞，处于等待状态，直到有数据到来，或者到了规定时间和出现错误为止。通过如下方法，能使 read()函数调用立即返回：

fcntl(fd,F_SETFL,FNDELAY);

FNDELAY 参数使 read()函数在端口没有字符存在的情况下，立即返回 0。如果要恢复到原来的没有数据就阻塞的状态，可以用下面的方法恢复：

fcntl(fd,F_SETFL,0);

7.2.3 设置串口属性

在前面已经对串口通信的一些概念和协议进行了相应介绍，明白了若要使用串口进行通信必须在通信两端设置好串口的属性，包括数据传输的波特率、传输的数据格式、是否有奇偶检验位、流控方式等。下面将介绍如何在嵌入式 Linux 平台下对这些串口通信属性进行设置。

若要设置串口属性，首先应在程序中包含 termios.h 的头文件，这个头文件包含终端控制结构和控制函数。在 termios.h 中有一个非常重要的数据结构 sructure termios，这个数据

结构包含了所有的串口参数，设置串口属性就是对这个数据结构赋值。在 termios.h 中还定义了两个重要的函数 tcgetattr()和 tcsetattr()，它们分别用于获取和设置串口的属性。它们的函数调用形式为 tcgetattr（fdcom, &termios_old）和 tcsetattr（fdcom, TCSANOW, &termios_new）。其中 fdcom 为串口的文件描述符，termios_old 和 termios_new 是 sructure termios 形式的数据结构，用于存放获得的串口属性和欲设置的串口属性。TCSANOW 表示设置立即生效，无须等到数据发送或接收结束。

1. 设置波特率

波特率是串口的通信速率，有输入和输出两个方向。常用的波特率参数如下：

波特率：	宏定义
2400 baud:	B2400;
4800 baud:	B4800;
9600 baud:	B9600;
19200 baud:	B19200;
38400 baud:	B38400;
57600 baud:	B57600;
115200 baud:	B115200;

其中，默认值为 9600 波特。

设置波特率可以使用 cfsetispeed(&termios_new, baudrate)和 cfsetospeed(&termios_new, baudrate)两个函数，参数 baudrate 是波特率的宏定义，如 B2400 就代表 2400baud。这两个函数分别设置入口端和出口端的速率，它们也是通过改变 struct termios 结构体的值实现的，由于在不同的操作系统中对其改变位置不一样，最好不要直接改变 termios 的值。

注意：通信两端的通信速率必须设为一致才能进行通信。在 temios 结构中有一个重要的成员 c_cflag，对串口属性的设置主要是对它的不同位进行“与”和“或”的操作。与速率相关的两个标志是：CLOCAL 和 CREAD。这两个参数必须保持使能状态，以确保程序在突发的作业控制或挂起时，不会成为端口的占有者，同时串口的接收驱动会自动读入数据。

2. 设置控制模式

（1）设置流控。

流控的具体含义及作用已作过介绍，流控的控制方式包括不使用流控、使用硬件流控、使用软件流控 3 种方式。具体实现是通过与 c_cflag 和 c_iflag 两个成员变量进行逻辑操作实现的。

- 若不使用流控：termios_new.c_cflag &= ~CRTSCTS。
- 使用硬件流控：termios_new.c_cflag |= CRTSCTS。
- 使用软件流控：termios_new.c_iflag |= IXON | IXOFF |IXANY。

（2）设置字符大小。

设置字符大小包括屏蔽字符大小位和设置字符数据位的大小，字符的大小是按位指定



的。设置方式如下：

```
termios_new.c_cflag &= ~CSIZE; //屏蔽字符大小位
termios_new.c_cflag |= CS8;        //选择 8 位数据位
termios_new.c_cflag |= CS7;        //选择 7 位数据位
termios_new.c_cflag |= CS6;        //选择 6 位数据位
termios_new.c_cflag |= CS5;        //选择 5 位数据位
```

（3）设置奇偶检验方式。

奇偶校验分为无校验、奇校验、偶检验 3 种方式，也是通过设置 c_cflag 标志位实现的。具体实现方法为：

```
无校验位：    termios_new.c_cflag &= ~PARENB;
奇校验：      termios_new.c_cflag |= PARENB;
              termios_new.c_cflag &= PARODD;
偶校验：      termios_new.c_cflag |= PARENB;
              termios_new.c_cflag &= ~PARODD;
```

以上是一些主要的串口控制模式，在串口通信中，还有一些属性需要设置，具体请参看后面的 7.3 节。

7.3 嵌入式 Linux 串口通信实例

在本章前面的章节中，已经介绍了串口通信的基本概念，以及在嵌入式 Linux 平台下串口访问 API 函数，在本节将以一个实例说明如何编写嵌入式 Linux 串口通信程序。

这个例程由 3 个部分组成：一部分是用于定义的头文件 MyCom.h；另一部分是实现串口操作的各个函数；还有一部分是测试程序 main()主函数，后两部分都保存在文件 MyCom.c 中。

其中 MyCom.h 定义了一个保存串口设置的数据结构 struct portinfo_t，还有对串口进行操作的各个函数的定义，如打开、关闭、接收和发送数据、设置串口属性等操作。

MyCom.c 中是对这些操作函数的实现，它首先要求对 struct portinfo_t 赋好值，然后依照 portinfo_t->tty 的值，利用函数 get_ptty()获得端口的名称，再利用端口名称打开相应的端口，并获得文件描述符，以后就可以进行串口属性设置和输入输出操作了，操作完毕要求用 PortClose()关闭端口。

以下为该实例的具体代码：

- 头文件 MyCom.h（见光盘/demo/chap07/7-2）

```
/*本程序符合 GPL 条约
MyCom.h
一组操作串口的函数
*/
```

```
//串口结构
typedef struct{
    char    prompt;         //prompt after reciving data
    int     baudrate;       //baudrate
    char    databit;        //data bits, 5, 6, 7, 8
    char    debug;          //debug mode, 0: none, 1: debug
    char    echo;           //echo mode, 0: none, 1: echo
    char    fctl;           //flow control, 0: none, 1: hardware, 2: software
    char    tty;            //tty: 0, 1, 2, 3, 4, 5, 6, 7
    char    parity;         //parity 0: none, 1: odd, 2: even
    char    stopbit;        //stop bits, 1, 2
    const int reserved;  //reserved, must be zero
}portinfo_t;
typedef portinfo_t *pportinfo_t;
/*
 * 打开串口，返回文件描述符
 * pportinfo：待设置的串口信息
*/
int PortOpen(pportinfo_t pportinfo);
/*
 * 设置串口
 * fdcom：串口文件描述符，pportinfo：待设置的串口信息
*/
int PortSet(int fdcom, const pportinfo_t pportinfo);
/*
 * 关闭串口
 * fdcom：串口文件描述符
*/
void PortClose(int fdcom);
/*
 * 发送数据
 * fdcom：串口描述符，data：待发送数据，datalen：数据长度
 * 返回实际发送长度
*/
int PortSend(int fdcom, char *data, int datalen);
/*
 * 接收数据
 * fdcom：串口描述符，data：接收缓冲区，datalen.：接收长度，baudrate：波特率
 * 返回实际读入的长度
*/
int PortRecv(int fdcom, char *data, int datalen, int baudrate);
```



- MyCom.c 文件

以下是文件 MyCom.c 的内容，这是程序的主体部分，包括串口操作的实现部分和测试代码。它要实现的功能是用参数 0 代表发送数据，用参数 1 代表接收数据。若是发送数据，它发送 100 次，每隔一秒发送一次，每次发送"1234567890"10 个字符，并依次打印发送的内容。若是接收数据，也是每次接收 10 个字符，每隔两秒钟接收一次，并打印各个接收到的字符和一次接收的数据长度，如果出错，就打印出错信息，它的代码如下：

```
/*本程序符合 GPL 条约
MyCom.c
*/
#include <stdio.h>              // printf
#include <fcntl.h>              // open
#include <string.h>             // bzero
#include <stdlib.h>             // exit
#include <sys/times.h>          // times
#include <sys/types.h>          // pid_t
#include <termios.h>            //termios, tcgetattr(), tcsetattr()
#include <unistd.h>
#include <sys/ioctl.h>          // ioctl
#include "MyCom.h"

#define   TTY_DEV "/dev/ttyS" //端口路径

#define TIMEOUT_SEC(buflen,baud) (buflen*20/baud+2)   //接收超时
#define TIMEOUT_USEC 0
/*******************************************
 * 获得端口名称
*******************************************/
char *get_ptty(pportinfo_t pportinfo)
{
        char *ptty;

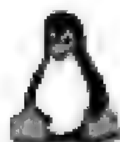
        switch(pportinfo->tty){
                case '0':{
                        ptty = TTY_DEV"0";
                }break;
                case '1':{
                        ptty = TTY_DEV"1";
                }break;
                case '2':{
                        ptty = TTY_DEV"2";
```

```
            }break;
    }
    return(ptty);
}

/*******************************************
 * 波特率转换函数
*******************************************/
int convbaud(unsigned long int baudrate)
{
    switch(baudrate){
        case 2400:
            return B2400;
        case 4800:
            return B4800;
        case 9600:
            return B9600;
        case 19200:
            return B19200;
        case 38400:
            return B38400;
        case 57600:
            return B57600;
        case 115200:
            return B115200;
        default:
            return B9600;
    }
}

/*******************************************
 * Setup comm attr
 * fdcom: 串口文件描述符, pportinfo: 待设置的端口信息
 *
*******************************************/
int PortSet(int fdcom, const pportinfo_t pportinfo)
{
    struct termios termios_old, termios_new;
    int       baudrate, tmp;
    char      databit, stopbit, parity, fctl;

    bzero(&termios_old, sizeof(termios_old));
```




```
    bzero(&termios_new, sizeof(termios_new));
    cfmakeraw(&termios_new);
    tcgetattr(fdcom, &termios_old);              //get the serial port attributions
    /*------------设置端口属性------------*/
    //baudrates
    baudrate = convbaud(pportinfo -> baudrate);
    cfsetispeed(&termios_new, baudrate);         //填入串口输入端的波特率
    cfsetospeed(&termios_new, baudrate);         //填入串口输出端的波特率
    termios_new.c_cflag |= CLOCAL;               //控制模式，保证程序不会成为端口的
占有者
    termios_new.c_cflag |= CREAD;                //控制模式，使能端口读取输入的数据

    // 控制模式，flow control
    fctl = pportinfo-> fctl;
    switch(fctl){
        case '0':{
            termios_new.c_cflag &= ~CRTSCTS;          //no flow control
        }break;
        case '1':{
            termios_new.c_cflag |= CRTSCTS;           //hardware flow control
        }break;
        case '2':{
            termios_new.c_iflag |= IXON | IXOFF |IXANY;  //software flow control
        }break;
    }

    //控制模式，data bits
    termios_new.c_cflag &= ~CSIZE;           //控制模式，屏蔽字符大小位
    databit = pportinfo -> databit;
    switch(databit){
        case '5':
            termios_new.c_cflag |= CS5;
        case '6':
            termios_new.c_cflag |= CS6;
        case '7':
            termios_new.c_cflag |= CS7;
        default:
            termios_new.c_cflag |= CS8;
    }

    //控制模式 parity check
    parity = pportinfo -> parity;
```

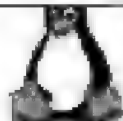
```
    switch(parity){
        case '0':{
            termios_new.c_cflag &= ~PARENB;        //no parity check
        }break;
        case '1':{
            termios_new.c_cflag |= PARENB;         //odd check
            termios_new.c_cflag &= ~PARODD;
        }break;
        case '2':{
            termios_new.c_cflag |= PARENB;         //even check
            termios_new.c_cflag |= PARODD;
        }break;
    }

    //控制模式, stop bits
    stopbit = pportinfo -> stopbit;
    if(stopbit == '2'){
        termios_new.c_cflag |= CSTOPB;    //2 stop bits
    }
    else{
        termios_new.c_cflag &= ~CSTOPB;  //1 stop bits
    }

    //other attributions default
    termios_new.c_oflag &= ~OPOST;            //输出模式, 原始数据输出
    termios_new.c_cc[VMIN]   = 1;             //控制字符, 所要读取字符的最小数量
    termios_new.c_cc[VTIME] = 1;              //控制字符, 读取第一个字符的等待时间
unit: (1/10)second

    tcflush(fdcom, TCIFLUSH);                 //溢出的数据可以接收, 但不读
    tmp = tcsetattr(fdcom, TCSANOW, &termios_new);   //设置新属性, TCSANOW: 所
有改变立即生效
    tcgetattr(fdcom, &termios_old);
    return(tmp);
}

/*******************************************
 * Open serial port
 * tty: 端口号 ttyS0, ttyS1, ....
 * 返回值为串口文件描述符
*******************************************/
int PortOpen(pportinfo_t pportinfo)
```




```
{
        int fdcom;      //串口文件描述符
        char *ptty;

        ptty = get_ptty(pportinfo);
        //fdcom = open(ptty, O_RDWR | O_NOCTTY | O_NONBLOCK | O_NDELAY);
        fdcom = open(ptty, O_RDWR | O_NOCTTY | O_NONBLOCK);

        return (fdcom);
}

/*******************************************
 * Close serial port
 *******************************************/
void PortClose(int fdcom)
{
        close(fdcom);
}

/*******************************************
 * send data
 * fdcom: 串口描述符, data: 待发送数据, datalen: 数据长度
 * 返回实际发送长度
 *******************************************/
int PortSend(int fdcom, char *data, int datalen)
{
        int len = 0;

        len = write(fdcom, data, datalen);      //实际写入的长度
        if(len == datalen){
                return (len);
        }
        else{
                tcflush(fdcom, TCOFLUSH);
                return -1;
        }
}

/*******************************************
 * receive data
 * 返回实际读入的字节数
 *
```

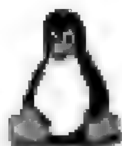
```
*********************************************/
int PortRecv(int fdcom, char *data, int datalen, int baudrate)
{
        int readlen, fs_sel;
        fd_set    fs_read;
        struct timeval tv_timeout;

        FD_ZERO(&fs_read);
        FD_SET(fdcom, &fs_read);
        tv_timeout.tv_sec = TIMEOUT_SEC(datalen, baudrate);
        tv_timeout.tv_usec = TIMEOUT_USEC;

        fs_sel = select(fdcom+1, &fs_read, NULL, NULL, &tv_timeout);
        if(fs_sel){
              readlen = read(fdcom, data, datalen);
              return(readlen);
        }
        else{
              return(-1);
        }

        return (readlen);
}

//************************Test*********************************
int main(int argc, char *argv[])
{
        int fdcom, i, SendLen, RecvLen;
        struct termios termios_cur;
        char RecvBuf[10];
        portinfo_t portinfo ={
              '0',                              // print prompt after receiving
              115200,                           // baudrate: 9600
              '8',                              // databit: 8
              '0',                              // debug: off
              '0',                              // echo: off
              '2',                              // flow control: software
              '0',                              // default tty: COM1
              '0',                              // parity: none
              '1',                              // stopbit: 1
              0                                 // reserved
        };
```




```
if(argc != 2){
        printf("Usage: <type 0 -- send 1 -- receive>\n");
        printf("    eg:");
        printf("            MyPort 0");
        exit(-1);
}

fdcom = PortOpen(&portinfo);
if(fdcom<0){
        printf("Error: open serial port error.\n");
        exit(1);
}

PortSet(fdcom, &portinfo);

if(atoi(argv[1]) == 0){
        //send data
        for(i=0; i<100; i++){
                SendLen = PortSend(fdcom, "1234567890", 10);
                if(SendLen>0){
                        printf("No %d send %d data 1234567890.\n", i, SendLen);
                }
                else{
                        printf("Error: send failed.\n");
                }
                sleep(1);
        }
        PortClose(fdcom);
}
else{
        for(;;){
                RecvLen = PortRecv(fdcom, RecvBuf, 10, portinfo.baudrate);
                if(RecvLen>0){
                        for(i=0; i<RecvLen; i++){
                                printf("Receive data No %d is %x.\n", i, RecvBuf[i]);
                        }
                        printf("Total frame length is %d.\n", RecvLen);
                }
                else{
                        printf("Error: receive error.\n");
                }
```

```
                sleep(2);
            }
        }
        return 0;
}
```

7.4 小结

从本章开始将介绍操作系统的外部扩展应用专题。本章主要介绍的是如何在嵌入式 Linux 下利用串口进行通信。

在嵌入式 Linux 下实现串口通信前，必须掌握串口通信的相关知识。在本章中重点介绍了同步通信与异步通信的特点与区别、波特率的概念、如何进行串口速率流控和差错控制，以及串口通信过程和 RS-232 协议等内容，这是进行串口开发的基础，应当重点掌握。只有这样才能解决开发中遇到的实际问题。

在嵌入式应用系统开发中，为自己开发的设备编写驱动程序是很频繁的事情。所以在本章中，通过实例介绍了在 Linux 下编写设备驱动程序的方法和其开发流程。

本章中通过一个完整的 Linux 下串口通信的例程，介绍了如何通过调用 API 函数实现对串口的访问和设置串口属性。

7.5 思考题

1．同步通信和异步通信的特点是什么？串口通信为什么一般要采用异步通信？

2．串口传输的速率用什么表示？它们表示的含义各是什么？它们有何关系？

3．串口的流控有几种方式？它们是如何实现的？

4．什么是 DTE？什么是 DCE？它们的通信过程是怎样的？

5．在 RS-232 中什么表示逻辑正？什么表示逻辑负？

6．试简述 DB-9 和 DB-25 各信号线功能。

7．字符型设备驱动程序注册各个入口函数是如何实现的？

8．请编写一个串口通信程序，实现以下功能：使用硬件流控，8 位字符大小，以 9600 的波特率从串口 1 发送键盘输入的字符串，从串口 2 接收，并在屏幕上打印出接收到的字符（提示：可以用一根串口自环线连接串口 1 和串口 2，串口 1 为输出端，串口 2 为输入端，具体实现请参考 http://metalab.unc.edu/pub/Linux/docs/LDP/programmers-guide/lpg-0.4.examples.tar.gz 的 miniterm 例程）。

第三篇 应 用 篇

第8章 键盘开发和时钟管理

知识点：

- 在嵌入式系统中扩展按键开关
- 设计一个矩阵键盘
- 编写自己的键盘驱动
- 嵌入式 Linux 对计数——定时器的管理
- 定时器的使用
- 获取和设置系统日期

本章导读：

本章将对在嵌入式 Linux 系统中扩展键盘及使用计数——定时器进行说明。首先，从最简单的键盘（按键开关）出发，介绍了如何通过外部的按键开关控制嵌入式系统的运作，在嵌入式 Linux 系统中如何访问外部的开关端口信息；然后，介绍了如何在自己的系统中扩展一个相对复杂的矩阵键盘，设计一个矩阵键盘应该注意的问题，以及如何编写自己的键盘驱动程序；最后，介绍了嵌入式 Linux 的时钟管理的相关知识，主要讲述如何使用系统提供的时钟设置自己的定时器，以及如何获取系统的日期和时间，如何设置系统的日期和时间等。

8.1 最简单的键盘——按键开关

在嵌入式系统中，许多时候系统的运行都需要操作人员的干涉，比如POS机、微波炉、复印机、传真机等，都需要操作者发出一些指令，告诉机器该如何做，这就要求系统必须提供人机交互接口。键盘作为一种最常用的输入工具，不但在PC中是标准配置，在许多嵌入式系统也是需要的。

但是，嵌入式系统往往是针对具体的应用而设计的，各种应用对输入设备的要求也各不相同。如有时需要外接一个标准的PC机键盘（如工控机）；有时需要一个小键盘就可以了（如POS机）；有时可能只需要一个按键开关即可（如电饭煲）。开发者应当能够根据系统的具体需要在嵌入式系统中扩展自己的键盘。下面将讲述如何在嵌入式Linux系统中使用最简单的键盘——按键开关。

8.1.1 按键开关电路

按键开关通常是指通过外力使电路瞬时接通的开关，在许多场合都有应用。比如大多数处理器的RESET电路都用到了按键开关，它通过按键产生一个瞬时的低电压，CPU感知这个低电压后重启。在有些系统中也用按键开关切换工作模式，它通过按键开关生成一个低压脉冲，产生一次中断，在中断处理程序中改变工作模式，并且通过置外部标志的方式告知用户当前的工作模式，通过切换开关，就可以实现在不同工作模式之间进行切换。

通常的标准键盘也是由许多按键开关组成的。其中，每个按键开关的电路示意图如图8-1所示。

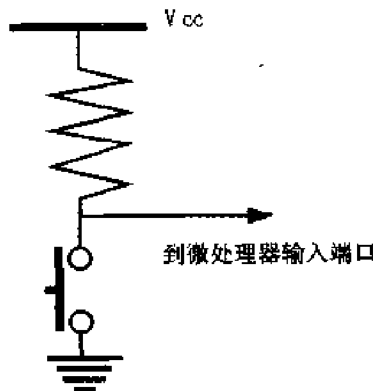
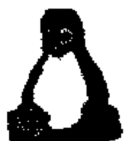


图8-1 按键开关电路示意图

当开关打开时，输出高电压，为逻辑1；当按键按下时，电平输出点与地相连，输出低电平，为逻辑0。



8.1.2 去除按键抖动

如图 8-1 所示的按键开关的电路是最简单的，遗憾的是，它并不完善，因为它按下或者被释放时，并不能明确地产生一个逻辑 0 或者逻辑 1。由于按键是机械触点，当机械触点断开、闭合时，会产生抖动。

这种抖动对于用户来说是感觉不到的，但对计算机来说，则是完全可以感应的（因为计算机处理的速度是在微秒级，而机械抖动的时间至少是毫秒级），这对计算机而言，已是一个“漫长”的时间了。假如利用按键开关产生中断可能就会产生一个问题，就是说按键有时灵，有时不灵，其实就是这个原因，有可能只按了一次按键，可是计算机却已执行了多次中断的操作。

图 8-1 所示电路输出的波形如图 8-2 所示。

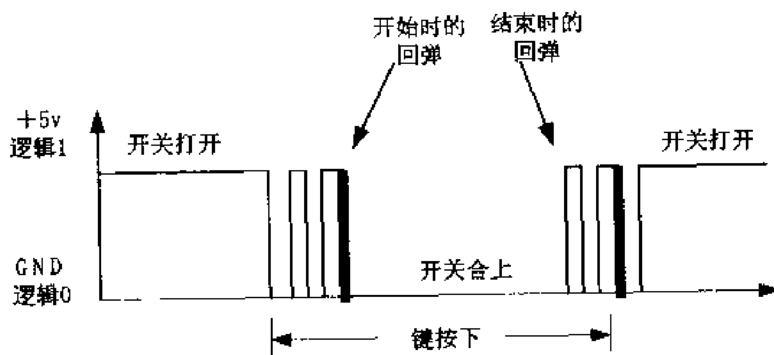


图 8-2 按键波形图

可以看出，在按键开始和结束时，输出的波形并不是简单的低电平或高电平，它存在着时上时下的抖动。

为使 CPU 能正确地读出按键的状态，对每一次按键只作一次响应，就必须考虑如何去除抖动。常用的去除抖动的方法有两种：软件方式和硬件方式。

对于简单的按键电路，可以采用软件方法去除抖动。软件法其实很简单，就是在程序获得外接端口为低的信息后，不是立即认定按键已被按下，而是延时 10 毫秒或更长一些时间后再次检测外部端口，如果仍为低，说明按键的确按下了，这实际上是避开了按键按下时的抖动时间。同理，在检测到按键释放后再延时 5~10 个毫秒，消除后沿的抖动，然后再对键值处理。实践证明，不对按键释放的后沿进行处理，通常也能满足一定的要求。

但有时用软件方式并不能很好地解决按键抖动问题，例如按键开关连接的是中断请求线，程序是不能读取中断请求线的状态的，这时就需要使用硬件方法。硬件方法其实就是一个去除抖动电路，它用于把按键和放键时的抖动波形去掉，这个电路也是比较简单的，读者可以查找相关去抖电路。对于比较复杂的矩阵键盘而言，通常使用去除抖动的芯片去除抖动，例如键盘接口芯片 8279、MAX6816/MAX6817/MAX6818 等。

8.1.3 把按键接入嵌入式系统

把按键开关接入嵌入式应用系统中，使系统能够感知按键状态的变化的方法有轮询和中断。

轮询方式通过把按键开关直接连接到系统外部 I/O 总线上，使程序以访问外部端口的方式获知总线状态，然后再读取按键开关所连接的位，从而判断出开关的状态。程序才不断地读入外部端口的数值，如果有改变，就可以判断按键已经被按下或者被放开（当然如果使用软件去除抖动的方式的话，应该延缓一段时间再读一次以确认）。

注意：*采用轮询方式对端口进行访问的间隔必须足够短，要求小于通常一次按下的时间，否则就可能发生按键已经按下系统却不知道的情况。*

采用轮询方式的效率是非常低的，它只能用于一些比较简单且功能单一的应用系统中。在大多数情况下，都是使用中断方式。使用中断方式在嵌入式系统中扩展按键开关的示意图如 8-3 所示。

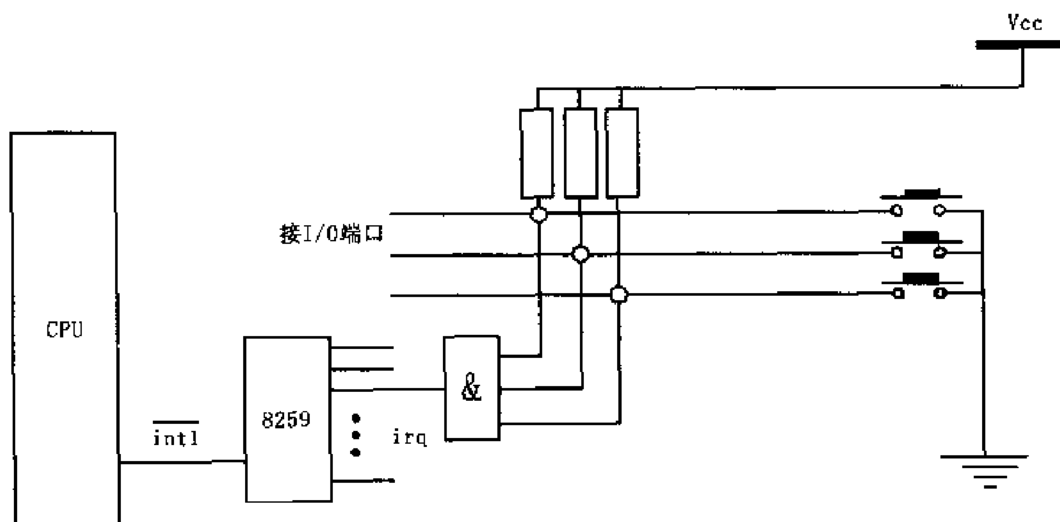


图 8-3　把按键开关接入嵌入式系统

在图 8-3 所示的示意图中，各个按键都接到一个与门上，当有任何一个按键按下时，都会使与门输出为低电平，从而引起中断，进入中断处理程序。在中断处理程序中，系统通过 I/O 端口读取按键状态，然后判断哪个按键被按下。它的好处是不用在主程序中不断地循环查询，如果有键按下，系统就会知道并进行相应处理。

8.2 在嵌入式系统中扩展键盘

在 8.1 节中讲述了如何在嵌入式系统中引入按键开关。在有些系统中，如果只使用简



单的按键开关作为输入还不能满足要求，这时就需要使用键盘，包括为系统特殊设计的小键盘或者标准键盘。下面将介绍如何在嵌入式系统中扩展自己的键盘。

8.2.1 矩阵键盘

要在嵌入式系统中按应用的特殊要求，扩展自己的小键盘，如果需要的键数比较多时（5 个键以上），采用矩阵法来做键盘是合理的。因为 CPU 的 I/O 线是很有限的，不可能在一个 I/O 线上只接一个按键。

矩阵键盘的连接方法如图 8-4 所示。在矩阵式键盘中，每条水平线和垂直线在交叉处都不直接连通，而是通过一个按键加以连接。这样，一个 8 位端口就可以构成 4×4=16 个按键，比直接将线用于键盘多出了一倍，而且线数越多，区别就越明显，比如再多加一条线就可以构成 20 键的键盘，而直接用 I/O 线则只能多出一键（9 键）。

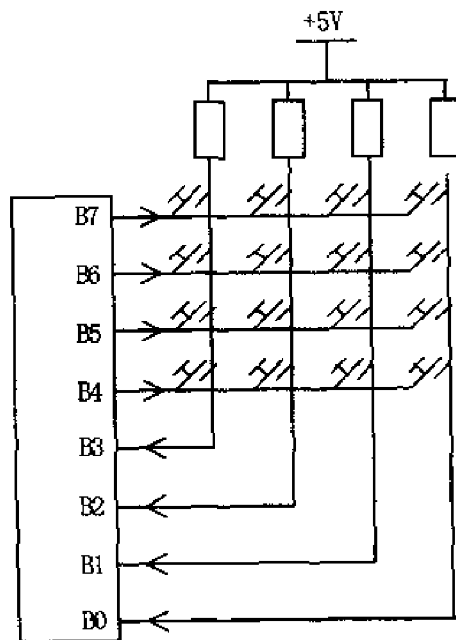
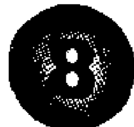


图 8-4　矩阵键盘连接图

矩阵式结构的键盘显然比直接法要复杂，其识别也要复杂一些。在图 8-4 中，列线通过电阻接正电源，并将行线所接的 I/O 口作为输出端，而列线所接的 I/O 口则作为输入端。这样，当按键没有按下时，所有的输出端都是高电平，代表无键按下。由于行线输出是低电平，一旦有键按下，则输入线（列线）就会被拉低，这样，通过读入输入线（列线）的状态就可得知是否有键按下了。具体的识别及编程方法如下所述。

如何确定矩阵式键盘上哪个键被按下，这里介绍一种"行扫描法"。行扫描法又称为逐行（或列）扫描查询法，是一种最常用的按键识别方法，扫描过程分为以下两步：

（1）判断键盘中有无键按下。将全部行线置低电平，然后检测列线的状态。只要有一



列的电平为低，则表示键盘中有键被按下，而且闭合的键位于低电平线与 4 根行线相交叉的 4 个按键之中。若所有列线均为高电平，则键盘中无键按下。

（2）判断闭合键所在的位置。在确认有键被按下后，即可进入确定具体闭合键的过程。其方法是：依次将行线置为低电平，即在置某根行线为低电平时，把其他行线置为高电平。在确定某根行线位置为低电平后，再逐行检测各列线的电平状态。若某列为低，则该列线与置为低电平的行线交叉处的按键即为闭合按键。

例如在图 8-4 中，CPU 的低 8 位用作键盘 I/O 口，其中，键盘的列线连接到 I/O 口的低 4 位，键盘的行线连接到 I/O 口的高 4 位。列线 B0~B3 分别连接有 4 个上拉电阻到正电源+5V，并把列线设置为输入线，行线 B4~B7 设置为输出线，4 根行线和 4 根列线形成 16 个相交点。如果进行键盘扫描，再加上去除抖动的功能，要执行如下操作：

（1）检测当前是否有键被按下。检测的方法是 B4~B7 输出全“0”，读取 B0~B3 的状态，若 B0~B3 为全“1”，则无键闭合，否则为 0 的那一列有键闭合。

（2）去除键抖动。当检测到有键按下后，延时一段时间再作下一步的检测判断。

（3）若有键被按下，应识别出是哪一个键闭合。方法是对键盘的行线进行扫描。B4~B7 按下述 4 种组合依次输出：

B7　1　1　1　0

B6　1　1　0　1

B5　1　0　1　1

B4　0　1　1　1

（4）在每组行输出时读取 B0~B3，若全为“1”，则表示设为“0”的这一行没有键闭合，否则有键闭合。由此得到闭合键的行值和列值，通常是一个扫描码，然后可采用计算法或查表法将闭合键的行值和列值转换成所定义的键值。为了保证键每闭合一次 CPU 仅作一次处理，必须去除按键释放时的抖动。

通过以上步骤就可以得到被按下的键的扫描码了，通常把扫描码放到一个缓冲区内，直到应用程序处理按键为止。缓冲是一个很有用的措施，因为当应用程序在按键发生不能处理它们时，通过缓冲区就可以防止按键的丢失。缓冲区的大小取决于应用程序的需要，一般应该大于 10。一般来说，都是把缓冲区作为一个环行队列来管理。使用两个指针，一个指向第一个空位，一个指向第一个扫描码。当一个按键被按下时，扫描码将被放置在环形队列的空指针指向的位置。而应用程序则是通过指向第一个扫描码的指针去读取扫描码。若缓冲区已满，则任何下一个按键都将被丢弃。

为了最大限度地利用按键的数量，可以在键盘中加入 Shift 键，规定 Shift 键打开时，一个按键被按下是一个值；若 Shift 键与按键被同时按下，这时表示的是另外一个值。这样就可以在按键不变的情况下扩充键值。一个使用两个 Shift 键的键盘电路如图 8-5 所示。

矩阵中的每一个键都有一个与之相关联的扫描码，当 Shift 键没有被按下时，键的扫描码在 0~15 之间。当 Shift1 被按下时，每个键的扫描码在对应值上加上 16；若 Shift2 被按下，则在对应值上加上 32；若 Shift1 和 Shift2 同时被按下，则加上 48，具体如表 8-1 所示。



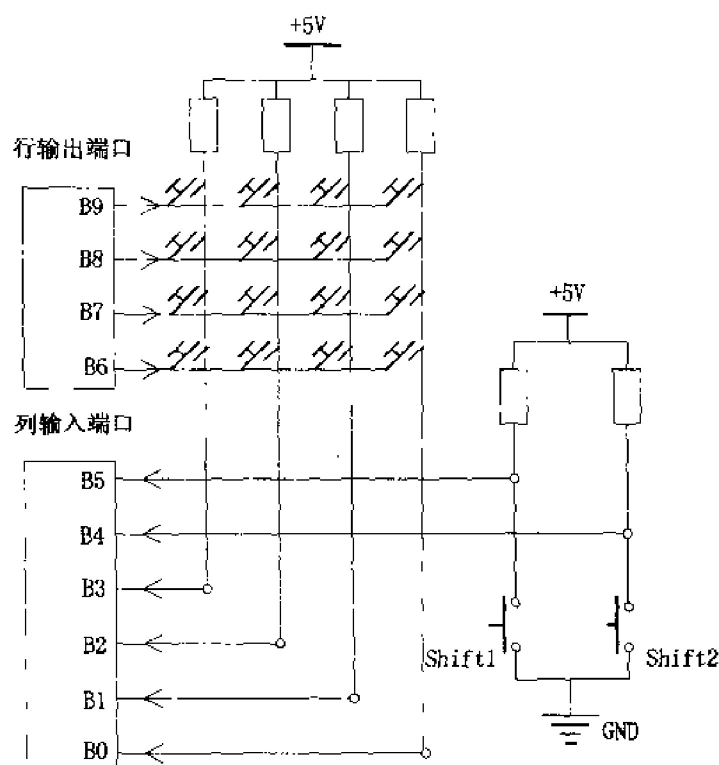


图 8-5 使用切换键的矩阵键盘

表 8-1 键盘扫描码

扫 描 代 码	被按下的 Shift 值
0~15	没有
16~31	Shift1
32~47	Shift2
48~63	Shift1 和 Shift2

8.2.2 用 Intel 8279 扩展键盘

Intel 8279 是一种通用的可编程序的键盘、显示接口器件，单片器件就能够完成键盘输入和显示控制两种功能。键盘部分提供一种扫描的工作方式，可以和具有 64 个按键的矩阵键盘相连接，能对键盘不断扫描，自动去除抖动，自动识别按下的键并给出编码，能对双键或 n 键同时按下实行保护。显示部分为发光二极管、荧光管及其他显示器提供了按扫描方式工作的显示接口，它为显示器提供多路复用信号，可以显示多达 16 位的字符或数字。

1. 结构框图

Intel 8279 内部又可以分为互相关联的各个部分，其中重要的部件有输入/输出控制及数据缓冲器、定时寄存器及定时控制、扫描计数器、回复缓冲器、键盘消抖控制、FIFO/传感

器 RAM、显示 RAM 等。这些部件的具体功能将在随后的内容中详细介绍，它们所组成的 Intel 8279 的框图如图 8-6 所示。

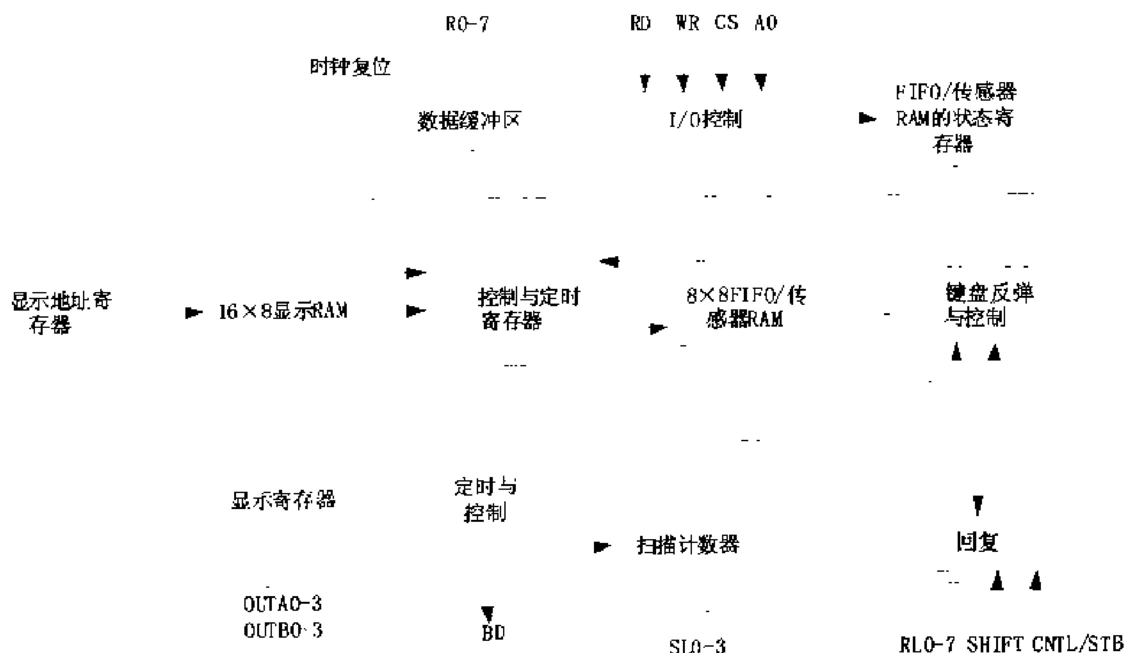


图 8-6　8279 体系结构框图

（1）输入/输出控制及数据缓冲器

数据缓冲器是双向缓冲器，连接内外部线，用于传送 CPU 和 8279 之间的命令或数据。其中用 A0 区别传送的信息的种类，若 A0=1，输入的是指令，输出的是状态字；若 A0=0，输入和输出的都是数据。

I/O 控制是 CPU 对 8279 进行控制的引线。CS 为片选信号，低电平有效，用 CPU 对 8279 的译码选中。WR 为写信号，RD 为读信号，都是低电平有效，在读或写之前必须由 CPU 置为低电平。

（2）控制与定时寄存器及定时控制

控制与定时寄存器用于寄存键盘及显示的工作方式，以及由 CPU 编程的其他操作方式。

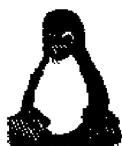
定时控制包括基本的计数链，首级计数器是一个可编程的 N 级计数器，N 可在 2~31 之间由软件控制，以便从外部时钟 CLK 得到内部所需要的 100kHz 时钟信号。然后，经过分频为键盘提供适当的逐行扫描频率和显示的扫描时间。

（3）扫描计数器

扫描计数器有两种工作方式。按编码方式工作时，计数器进行二进制计数，四位计数状态从扫描线 SL0-SL3 输出，经外部译码器译码后，为键盘和显示器提供扫描线；按译码方式工作时，扫描计数器的最低二位被译码后，从 SL0~SL3 输出。

（4）回复缓冲器、键盘去抖及控制

来自 RL0~RL78 根回复线的回复信号，由回复缓冲器缓冲并储存。



在键盘工作方式中，这些线被接到键盘矩阵的列线。在逐行扫描时，回复线用来搜索一行中闭合的键。当某一键闭合时，去抖电路就被置位，延时等待 100ms 之后，再检验该键是否连续保持闭合。若闭合，则该键的地址和附加的位移、控制状态一起形成键盘数据被送入 8279 内部的 FIFO 存储器。键盘的数据格式如下：

D_7	D_6	$D_5D_4D_3$	$D_2D_1D_0$
控制	移位	扫描	回复

控制和位移（D_7 和 D_6）的状态由两个独立的附加开关决定，而扫描（D_5、D_4、D_3）和回复（D_2、D_1、D_0）则是被按键的位置数据。D_5、D_4、D_3 三位来自扫描计数器，是按键的行编码，而 D_2、D_1、D_0 三位则是来自列计数器，它们是根据回复信号而确定的列编码。

在传感器矩阵方式中，回复线的内容直接被送往相应的传感器 RAM（即 FIFO 存储器）。而工作在选通输入方式时，回复线的内容在 CNTL/STB 线的脉冲上升沿时，被送入 FIFO 存储器。

（5）FIFO/传感器 RAM 及其状态

FIFO/传感器 RAM 是一个双重功能的 8×8RAM。在键盘或选通工作方式时，它是 FIFO 存储器。每次新的输入都顺序写入到 RAM 单元，而每次读出时，总是按输入的顺序，将最先输入的数据读出。FIFO 状态寄存器用于存放 FIFO RAM 的工作状态，例如 RAM 是满还是空；其中存有多少字符；是否操作出错等。当 FIFO 存储器不为空时，状态逻辑将产生 IRQ=1 信号，向 CPU 申请中断。

在传感器矩阵方式时，这个存储器又是传感器 RAM。它存放着传感器矩阵中每一个传感器的状态。在此方式中，若检索出传感器的变化，IRQ 信号将变为高电平，向 CPU 请求中断。

（6）显示 RAM 和显示地址寄存器

显示 RAM 用于存储显示数据。该区具有 16 个字节，也就是最多可以存储 16 个字节的显示信息。显示地址寄存器用于积存由 CPU 进行读/写的显示 RAM 的地址，它可以由命令设定，也可以设置成每次读出或写入之后自动递增。

2. Intel 8279 的引线

Intel 8279 采用 40 脚封装，管脚图如图 8-7 所示。

以下是各个引脚的功能说明：

- D7-D0（数据总线）：双向、三态总线。
- CLK（系统时钟）：输入。
- RESET（复位）：输入，高电平有效。复位时默认状态。
- CS（片选）：输入，低电平有效。
- A0（缓冲器地址）：输入。
- RD（读信号）和 WR（写信号）：输入，低电平有效。
- IRQ（中断请求）：输出，高电平有效。

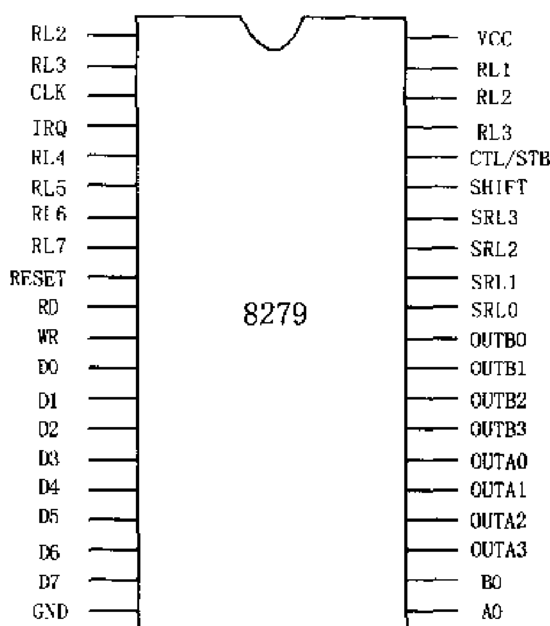


图 8-7 8279 引脚图

在键盘工作方式中，当 FIFO/传感器 RAM 存放有数据时，IRQ 为高电平。CPU 每次从 RAM 读出数据时，IRQ 就变为低电平。若 RAM 中仍有数据，则 IRQ 再次恢复为高电平。在传感器工作方式中，每当检测出传感器状态变化时，IRQ 就出现高电平。

- SL0~SL3（扫描线）：输出。
- RL0~RL7（回复线）：输入。它们是键盘矩阵或传感器矩阵的列信号输入线。
- SHIFT（换档信号）：输入，高电平有效。该信号线用于扩充键开关的功能，可以用作键盘的上、下档功能键。在传感器方式和选通方式中，SHIFT 无效。
- CNTL/STB（控制/选通）：输入，高电平有效。在键盘工作方式时，作为控制功能键使用。在选通方式时，该信号的上升沿可以将来自 RL0-RL7 的数据存入 FIFO 存储器。在传感器方式，无效。
- OUTA0~OUTA3（A 组显示信号）：输出。
- OUTB0~OUTB3（B 组显示信号）：输出。
- BD（消隐显示）：输出，低电平有效。该输出信号在数字切换显示或使用显示消隐命令时，将显示消隐。

3. Intel 8279 的命令和状态字

Intel 8279 是可编程接口芯片，可以通过编程使其实现相应的功能，编程过程实际上就是 CPU 向 8279 发送控制命令的过程。Intel 8279 共有 8 条命令和一个状态字寄存器。

（1）键盘/显示方式设置命令。

命令特征位：D7D6D5=000

0	0	0	D	D	K	K	K



DD 两位用来设定显示方式：

- 00：8 个字符显示……左入。
- 01：16 个字符显示……左入。
- 10：8 个字符显示……右入。
- 11：16 个字符显示……右入。

所谓的左入就是在显示时，显示字符是从左到右逐个排列。右入就是显示字符从右到左移动。所对应的 SL 编码最小的为显示的最高位。

KKK 三位用来设定键盘工作方式：

- K000：编码扫描键盘……双键锁定。
- K001：译码扫描键盘……双键锁定。
- K010：编码扫描键盘……n 键轮回。
- K011：译码扫描键盘……n 键轮回。
- K100：编码扫描传感器矩阵。
- K101：译码扫描传感器矩阵。
- K110：选通输入，编码显示扫描。
- K111：选通输入，译码显示扫描。

第一位 K 没有任何意义。双键锁定和 n 键轮回是两种不同的多键同时按下了保护方式。双键锁定为两键同时按下提供保护，在消振周期内，如果有两键同时被按下，则只有其中的一键弹起，而另一键在按下位置时，才能被认可。n 键轮回为 n 键同时按下提供保护，当有若干个键同时按下时，键盘扫描能根据发现它们的次序，依次将它们的状态送入 FIFO RAM。

（2）时钟编程命令。

命令特征位：D7D6D5=001

0	0	1	P	P	P	P	P

将来自 CLK 的外部时钟进行 PPPPP 分频（2-31）。

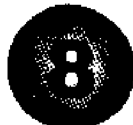
（3）读 FIFO/传感器 RAM 命令。

命令特征位：D7D6D5=010

0	1	0	AI	X	A	A	A

该命令字只在传感器方式时使用，在 CPU 读传感器 RAM 之前，必须用这条命令来设定将要读出的传感器 RAM 地址。由于传感器 RAM 的容量是 8×8bit，因此需要用命令字中的 3 位二进制代码 AAA 来选址。命令字中的 AI 为自动增量特征位，若 AI=1，则每次读出传感器 RAM 后，地址将自动增量（加 1），使地址指针指向顺序的下一个存储单元。这样，下一次读数便从下一个地址读出，而不必重新设置读 FIFO/传感器 RAM 命令。

在键盘工作方式中，由于读出操作严格按照先入先出的顺序，因此不必使用这条命令。



（4）读显示 RAM 命令。

命令特征位：D7D6D5=011

0	1	1	AI	A	A	A	A

在 CPU 读显示 RAM 之前，该命令字用来设定将要读出显示 RAM 的地址，四位二进制代码 AAAA 用于寻址显示 RAM 中的一个存储单元。如果自动增量特征位 AI=1，则每次读出后，地址自动加 1，使下一次读出顺序指向下一个地址。

（5）写显示 RAM 命令。

命令特征位：D7D6D5=100

1	0	0	AI	A	A	A	A

与前面命令字位含义相同，只是第一位不同。

（6）显示禁止写入/消隐命令。

命令特征位：D7D6D5=101

1	0	1	X	IW	IW	BL	BL

IW 用来掩蔽 A 组和 B 组（D3 对应 A 组，D2 对应 B 组）。例如，当 A 组的掩蔽位 D3=1 时，A 组的显示 RAM 禁止写入。因此，从 CPU 写入显示器 RAM 的数据不会影响 A 的显示，这种情况通常在采用双四位显示时使用。因为两个四位显示器是相互独立的，为了给其中一个四位显示器输入数据，而又不影响另一个四位显示器，因此必须对另一组的输入实行掩蔽。

BL 位是消隐特征，要消隐两组显示输出，必须设置两个 BL 位。若 BL=1，则执行此命令后，对应组的显示输出被消隐。若 BL=0，则恢复显示。

（7）清除命令。

命令特征位：D7D6D5=110

1	1	0	CD	CD	CD	CF	CA

该命令字用来清除 FIFO RAM 和显示 RAM。D4、D3、D2 三位（CD）用来设定清除显示 RAM 的方式，其意义如表 8-2 所示。

表 8-2　D4D3D2 表示的清除方式

D4	D3	D2	清 除 方 式
1	0	X	将显示 RAM 全部清“0”
1	1	0	将显示 RAM 置 20H（即 A 组=0010 B 组=0000）



（续）

D4	D3	D2	清 除 方 式
1	1	1	将显示 RAM 全部置 1
0			不清除（若 CA=1，则 D3、D2 仍有效）

D1（CF）位用来清空 FIFO 存储器。D1=1 时，执行清除命令后，FIFO RAM 被清空，使中断 IRQ 复位。同时，传感器 RAM 的读出地址也被清 0。

D0（CA）位是总清的特征位，它兼有 CD 和 CF 两者的功效。在 CA=1 时，对显示 RAM 的清除方式由 D3D2 的编码决定。

清除显示 RAM 大约需要 100μs 的时间。在此期间，FIFO 状态字的最高位 Du=1，表示显示无效。CPU 不能向显示 RAM 写入数据。

（8）结束中断/错误方式设置命令。

命令特征位 D7D6D5=111

1	1	1	E	X	X	X	X

这个命令有两个不同的应用：

- 作为结束中断命令。在传感器工作方式中，每当传感器状态出现变化时，扫描检测电路将其新的状态写入传感器 RAM，并启动中断逻辑，使 IRQ 变高，向 CPU 请求中断。并且禁止写入传感器 RAM。此时，如传感器 RAM 读出地址的自动递增特征没有置位（AI=0），则中断请求 IRQ 在 CPU 第一次从传感器 RAM 读出数据时就被清除。若自动递增特征已置位（AI=1），则 CPU 对传感器 RAM 的读出并不能清除 IRQ，而必须通过给 8279 写入结束中断/错误方式设置命令才能使 IRQ 变低。因此，在传感器工作方式中，此命令用于结束传感器 RAM 的中断请求。
- 作为特定错误方式的设置命令。在 8279 已被设定为键盘扫描 n 键轮回方式以后，如果 CPU 给 8279 又写入结束中断/错误方式设置命令（E=1），则在 8279 的消振周期内，如果发现有多个键被同时按下，则 FIFO 状态字中的错误特征位 S/E 将置位，并产生中断请求信号和阻止写入 FIFO RAM。

错误特征位 S/E 在读出 FIFO 状态字时被读出，而在执行 CF=1 的清除命令时被复位。

（9）8279 的 FIFO 状态字。

8279 的 FIFO 状态字，主要用于键盘和选通工作方式，以指示 FIFO RAM 中的字符数和是否有错误发生，其字位意义如下：

Du	S/E	O	U	F	N	N	N

- Du：Du=1 显示无效。
- S/E：传感器信号结束/错误特征码。
- O：O=1 出现溢出错误。

- U：U=1 出现不足错误。
- F：F=1 表示 FIFO RAM 已满 。
- NNN：为 FIFO RAM 中的字符数 。

对 FIFO RAM 的操作可能出现两种错误：溢出和不足。当 FIFO RAM 已满时，若其他的键盘数据企图写入 FIFO RAM，则出现溢出错误，状态字位“O”被置位。当 FIFO RAM 已被清空时，若 CPU 还企图读出，则会出现“不足”的错误，状态字位“U”被置位。

对于状态字的 S/E 位，当 8279 工作在传感器工作方式时，若 S/E=1，表示传感器的最后一个传感信号已进入传感器 RAM。当 8279 工作在特殊错误方式时，若 S/E=1，表示出现了多键同时被按下的错误。

当显示 RAM 由于清除命令尚未完成时，这时对显示 RAM 的操作无效，这样状态字的最高位 Du 被置位。

4. 8279 连接图

一个利用 8279 扩展键盘的连接如图 8-8 所示。

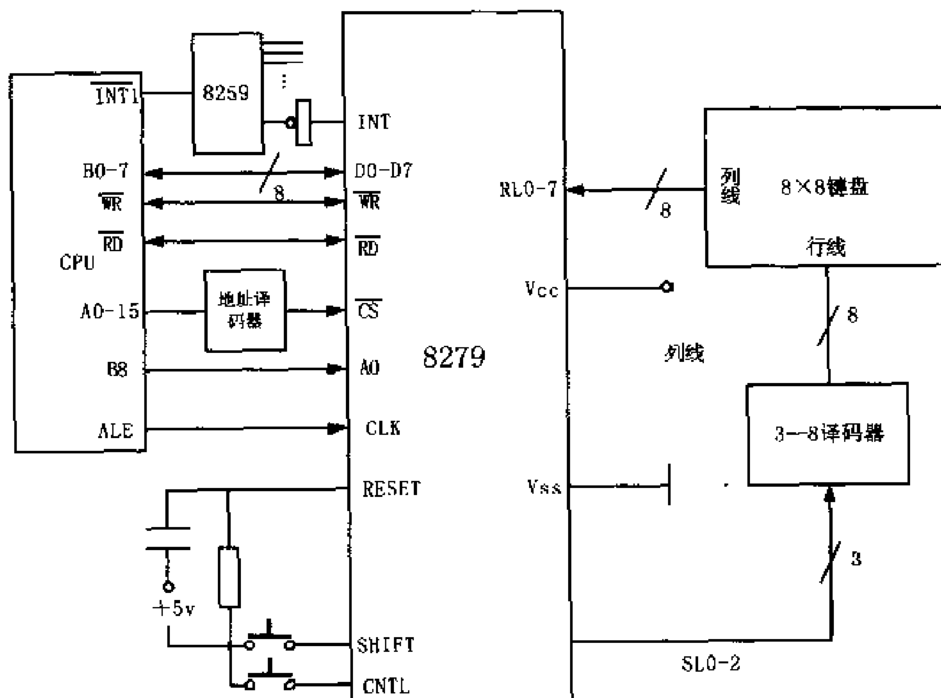


图 8-8　8279 与 CPU 的连接示意图

在图 8-8 中可以看到，8279 的数据总线与 CPU 的低 8 位相连，其连接是双向的，用于发送命令和读取 FIFO 状态（A0＝1）与传递数据（A0＝0）。而片选引脚 CS 和 CPU 的地址线经过地址译码器相连，所以在 CPU 给出 8279 的片选地址后，在地址译码器与 CS 相连的那根线的电平将为低，选中 8279。8279 的时钟引脚 CLK 与 CPU 的 ALE 引脚相连，由 CPU 提供时钟驱动，但是在 8279 内部又应该对 CPU 提供的时钟进行分频，以保证 100KHz 左右的扫描频率，分频范围为 0~32，在时钟编程命令中设置。8279 的 WR、RD



分别和 CPU 的 WR、RD 相连，分别表示对 8279 是写还是读，低电平有效。需要注意的是 8279 的中断请求引脚 INT，它是高电平有效。在键盘工作方式下，只要 FIFO 中还有扫描字符，INT 就为高，请求 CPU 进行中断处理。由于一般 CPU 的引脚是低电平有效，所以必须在它们之间加一个反向门。

至于 8279 与外部键盘的连接，8279 最多可以接一个 8×8 的矩阵键盘，它用 SL0~2 经过 3-8 译码器译成 8 根线作键盘行线，为扫描输入，而用 RL0~RL7 作列线，为键盘输出。扫描的原理在前面已经介绍过。

5. 一个键盘驱动程序实例

在前面部分已经对如何在一个嵌入式系统中扩展自己的小键盘以及相关电路连接进行了说明。而在嵌入式系统的软件部分，要做的就是把前面的硬件电路驱动起来，为它们写一个驱动程序，下面就以一个实例进行说明如何编写键盘驱动程序。

键盘作为 Linux 默认输入设备，一般嵌入式 Linux 操作系统都带有标准键盘的驱动程序。键盘是一个字符设备，它的驱动程序为 drivers/char/keyboard.c，其对应的头文件为 keyboard.h。在 Intel 架构中，键盘把自己固定在 IRQ1 上，键盘状态的寄存器地址为 0x64h，键盘扫描码的寄存器地址为 0x60h。当用户按下按键后，将产生中断，在中断处理程序中，将读出键盘状态和键盘扫描码，然后把键盘扫描码通过对应表转换为 ASC 码，送往用户应用程序。

本节的实例也是针对 Intel x86 体系结构。参照标准键盘的驱动程序，这个实例也使用 IRQ1 中断，从 0x64h 读入键盘状态，从 0x60h 读入键盘扫描码，这样就跟系统自带的键盘驱动程序发生冲突了，这时的新添驱动程序是无法捕获按键中断的。由于系统自带的驱动程序和要写的驱动代码不能共存，所以首先必须使系统自带的键盘驱动释放中断。但因为在内核源文件（drivers/char/keyboard.c）里它是作为一个静态符号被定义的，所以没有办法恢复它，只能重新启动，才能恢复原来的中断。

本节这段代码把它自己绑定到 IRQ1 上，在 Intert 结构下这是键盘控制的 IRQ。这时，当它连接到一个键盘中断时，将读出键盘的状态（这是 inb(0x64)的目的）和由键盘返回的扫描码。随后只要内核认为可以时，它将运行 got_char 给出键的编码（扫描码的前 7 位）和是否被按下的信息（如果第 8 位是 0 则表示按下，是 1 表示释放）。

下面是键盘驱动实例代码（见光盘/demo/chap08/8-1）：

```
/*键盘驱动范例   keybd.c*/
/* Copyright (C) 1998 by Ori Pomerantz */

/*必要头文件*/

/*标准头文件*/
#include <linux/kernel.h>     /*内核工作*/
#include <linux/module.h>    /*明确指定是模块*/

/*处理 CONFIG_MODVERSIONS */
```

```
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <linux/sched.h>
#include <linux/tqueue.h>

/* 在程序中将用到中断 */
#include <linux/interrupt.h>

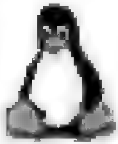
#include <asm/io.h>

/*在 2.2.3 版/usr/include/linux/version.h 中包含这个宏，
  但 2.0.35 版不包含，因此在此加入以备需要*/
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif

/*Bottom Half - 一旦内核模块认为它做任何事都是安全的时候，就将被内核调用*/
static void got_char(void *scancode)
{
  printk("Scan Code %x %s.\n",
    (int) *((char *) scancode) & 0x7F,
    *((char *) scancode) & 0x80 ? "Released" : "Pressed");
}

/*这个函数为键盘中断服务，它读取来自键盘的相关信息，然后安排到当内核认为 bottom
half 安全的时候让它运行*/
void irq_handler(int irq,
                 void *dev_id,
                 struct pt_regs *regs)
{
  /*这些变量是静态的，因为它们需要对 bottom half 可见（通过指针）*/
  static unsigned char scancode;
  static struct tq_struct task =
      {NULL, 0, got_char, &scancode};
  unsigned char status;

  /*读取键盘状态*/
  status = inb(0x64);
   /*读取扫描码*/
```




```
  scancode = inb(0x60);

  /*安排 bottom half 运行*/
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,2,0)
  queue_task(&task, &tq_immediate);
#else
  queue_task_irq(&task, &tq_immediate);
#endif
  mark_bh(IMMEDIATE_BH);
}

/*在这里假设驱动的加载仍是采用模块化方式*/
/*初始化模块--登记 IRQ 句柄*/
int init_module()
{
  /*由于原来键盘的句柄不能和本驱动程序共存，所以在启动本程序前不得不关闭它(释放它的 IRQ)。
  *同时因为不知道它在哪儿，所以换占以后没有办法恢复它，因此当本程序运行完后计算机将被重新启动
   */
  free_irq(1, NULL);

  /*请求 IRQ 1，键盘的 IRQ，指向我们的 irq_handler*/
  return request_irq(
    1,  /* PC 上的键盘的 IRQ 号*/
    irq_handler,  /* 我们的句柄*/
    SA_SHIRQ,
    /* SA_SHIRQ 意味着将这个 IRQ 指定为可以共享
     *
     * SA_INTERRUPT 能使句柄为一个快速中断
     */
    "test_keyboard_irq_handler", NULL);
}


/*消除*/
void cleanup_module()
{
  /*这段代码在此只是为了使驱动程序结构完整，它是完全不相关的，因为没有办法恢复被屏蔽的系统自带键盘中断，因此计算机完全没用了，需要被重新启动*/
  free_irq(1, NULL);
}
```

8.3 嵌入式 Linux 时钟管理

在许多基于微处理器的嵌入式系统中，对时间的管理是非常重要的，如在目前刚刚兴起的 PDA 手机中，不仅包含显示时钟功能，还包含定时提醒的记事本功能。在同样是嵌入式系统的摄像机中，也可以设定自动开机的时间。这些都包含时钟的管理功能。

在任何一个嵌入式系统中，都有一个硬件时钟，它相当于整个系统的心脏，给系统提供始终如一的时钟频率，是系统衡量其他一切时间的基准，也是系统进行时钟管理的基础。如果要提供日历功能，即使在系统关机时，也应该有后备电源维持这个时钟的跳动。

在嵌入式 Linux 系统中另外还有一个软件时钟，Linux 内核从硬件时钟上直接取得时间。在引导期间，Linux 将自己的软件时钟设置成与硬件时钟同步。在这以后，两个时钟都独立地运行。Linux 维持着自己的时钟，因为读取硬件时钟的速度很慢，并且其实现也比较复杂。

由于嵌入式 Linux 操作系统已经提供对时钟的管理，这样在开发牵涉到时钟方面的应用时，只需使用相应的系统调用就可以了。嵌入式 Linux 的时钟管理主要有两个方面：一个是设置和获取系统时间，另一个是使用定时器。下面将分别对这两个方面进行介绍。

8.3.1 时间日期管理

1. 时间的表示方式

在 Linux 操作系统中，有 3 类数据结构用来表示时间，它们都是在 time.h 中定义。

（1）日历时间表示法。

struct time_t 是一种最为紧凑的表达方式，用于表达日历时间，当它存储的是绝对时间时，表示的是自 1970 年 1 月 1 日 0 时（标准时区）起所经历的秒数。time_t 在数据结构上等于 long int。

（2）精确表示法。

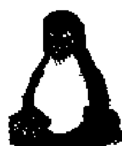
struct timeval：由于 struct time_t 的精度只有 1 秒，在对时间精度要求较高的场合将不能满足要求，所以定义了 struct timeval，它的数据结构为：

```
struct timeval{
long int tv_sec;/*秒数*/
long int tv_usec/*微秒数*/
}
```

所以 struct timeval 的时间精度为 1 微秒。

（3）详细表示法。

struct tm：在前面两种表示方式中，都将系统时间表示为相对于基准时间所经历的秒数，这对于计算是比较方便的，但是不符合人们日常表示时间的习惯，而且人们通常都看不懂。struct tm 把系统时间表示为年、月、日、时、分、秒这样人们容易接受的分散表示方式。



一个分散表示的数据结构总是和它所在的时区联系的，所以在这个数据结构中还应该说明它所在的时区，它的结构为：

```
struct tm{
int tm_sec;/*秒，0~61，之所以大于 59 是为了润秒*/
int tm_min; /*分，0~59*/
int tm_hour; /*时，0~23*/
int tm_day; /*天，1~31*/
int tm_mon; /*月，0~11*/
int tm_year; /*相对于 1900 年的年数*/
int tm_waday; /*相对于上一个星期天的天数，0~6*/
int tm_yday; /*相对于本年 1 月 1 日的天数，0~365*/
int tm_isdst; /*如果夏时制生效，则夏时制标志值为正；如果以非夏时制时间则为 0；如果此信息不可用，则为负*/
long int tm_gmtoff; /*标准时间换算为本地时间应该增加的秒数*/
const char *tm_zone; /*时区信息，只用于BSD 等增强版本中，一般不可用*/
}
```

2. 对时间操作的系统函数

（1）time_t time(time_t *result)

time_t time(time_t *result)函数用于返回系统相对于 1970 年 1 月 1 日 0 时的秒数。如果 result 不是一个 NULL 指针的话，也把结果赋给 result 指针指向的变量。如果日历时钟不可用，则返回-1。

（2）int gettimeofday (struct timeval *tp, struct timezone *tzp)

int gettimeofday (struct timeval *tp, struct timezone *tzp)函数用于得到当前日期和时间，并把它放在 timeval 结构的变量 tp 中，关于时区的信息则存储在 tzp 指向的数据结构中，如果 tzp 为 NULL，则忽略时区。

如果函数调用成功，返回 0；如果失败，则返回-1。

（3）int settimeofday(const struct timeval *tp,const struct timezone 1)*tzp)

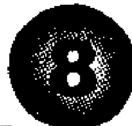
int settimeofday(const struct timeval *tp,const struct timezone *tzp)函数用于设置系统时间，即把系统时间改为 tp 指向的时间。tzp 中存储的为欲设置的时区信息，如果没有，可以设为 NULL。

如果函数调用成功，返回 0；如果失败，则返回-1。

（4）int adjtime(const struct timeval *delta,struct timeval *olddelta)

int adjtime(const struct timeval *delta,struct timeval *olddelta)函数用于调整系统时间。它是相对于系统当前时间而进行调整。其中 delta 表示要调整的时间数，若为正，则系统时间向前加速；若为负，则系统时间向后减速。如果 olddelta 不是 NULL，则函数用它来存储系统调整前的时间。

若调用成功，则返回 0，否则返回-1。



（5）struct tm *localtime(const time_t *time)

struct tm *localtime(const time_t *time)函数用于把 time_t 结构表示的时间转换为 tm 结构表示的时间，并表示为本地时间。调用这个函数还会把系统的时区信息写到 tzname 中。

（6）struct tm *gmtime(const time_t *time)

struct tm *gmtime(const time_t *time)函数的功能与上一个函数的功能差不多，只是它转换为格林威治时间。

（7）time_t mktime(struct tm *brokentime)

time_t mktime(struct tm *brokentime)函数用于把 struct tm 结构的时间转换为以 time_t 表示的时间。同时，该函数还会改变 brokentime 下的 tm_wday 和 tm_yday 的值，把它们改为基于其他时间基准的值。

如果转换失败，则返回-1，并且不会改变 brokentime 的值。

（8）Char * asctime(const struct tm* brokentime)

Char * asctime(const struct tm* brokentime)函数用于把用 struct tm 表示的时间转换为一串 ASC 字符串，例如 "Tue May 21 13:46:22 1991\n"。

（9）char * ctime(const time_t *time)

char*ctime(const time_t *time)用于把用 time_t 数据结构表示的时间转换为一串 ASC 字符串。

（10）size_t strftime (char *buf, size_t maxsize, const char 。1)*format,const struct tm *brokentime)

最后一个参数 struct tm *brokentime 是要格式化的时间值，由一个指向一个年、月、日、时、分、秒、周日时间值的指针加以说明。格式化结果存放在一个长度为 maxsize 个字符的 buf 数组中，如果 buf 长度足以存放格式化结果及一个 null 终止符，则该函数将返回在 buf 中存放的字符数（不包括 null 终止符），否则该函数返回 0。

参数 format 用于控制时间值的格式。如同 printf()函数一样，变换说明的形式是百分号之后跟一个特定字符。format 中的其他字符则按原样输出。两个连续的百分号在输出中产生一个百分号。与 printf()函数的不同之处是，每个变换说明都将产生一个定长输出字符串，在 format 字符串中没有字段宽度修饰符。表 8-3 中列出了 21 种 ANSI C 规定的变换说明。

表 8-3 strftime 输出格式转换说明

格　式	说　明	例　子
%a	缩写的周日名	Tue
%b	缩写的月名	Jan
%A	全周日名	Tuesday
%B	月全名	January
%c	日期和时间	Tue Jan 14 19:40:30 1992
%d	月日：[01, 31]	14
%H	小时（每天 24 小时）：[00, 23]	19
%I	小时（上、下午各 12 小时）：[01, 12]	07



（续）

格　式	说　明	例　子
%j	年日：[001, 366]	014
%m	月：[01, 12]	01
%M	分：[00, 59]	40
%p	AM/PM	PM
%S	秒：[00, 61]	30
%U	星期日周数：[00, 53]	02
%w	周日：[0=星期日，6]	2
%W	星期一周数：[00, 53]	02
%x	日期	01/14/92
%X	时间	19:40:30
%y	不带公元的年：[00, 991]	92
%Y	带公元的年	1992
%Z	时区名	MST

以上各个函数的关系如图 8-9 所示。

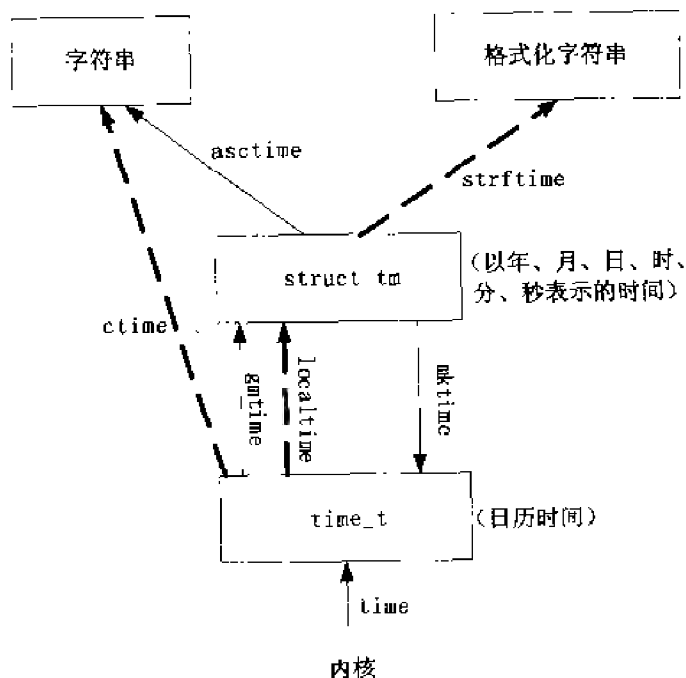


图 8-9　各时间函数的关系

3．实例

下面将通过一个对时间加以操作的实例来对上面所介绍的时间函数的应用加以说明。在该实例中，首先得到系统的时间，并将其转换成字符串形式打印出来。然后，执行一个复杂度比较高的运算，计算出这次运算的时间。其实现代码如下（见光盘/demo/chap08/8-2）：

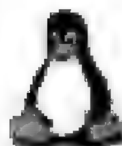
```
    /*时间/日期管理实例*/
    #include < time.h>
#include <stdio.h>
#include <math.h>

/*耗时运算函数*/
void function()
{
        unsigned int i,j;
        double y;
        for(i=0;i<5000;i++)
        for(j=0;j<4000;j++)
          y=sin((double)i);
}

main()
{
        time_t t;
        struct tm nowtime ,*pt;
        char *sTime;
        struct timeval tpstart,tpend;
          float  timeuse;

        /*获取系统时间*/
        t=time(NULL);
        pt=&nowtime;
        /*把 time_t 结构的时间表示为 tm 结构*/
        pt=localtime(&t);
        /*转换为字符串*/
        sTime=asctime(pt);
        printf("Now time is %s\n",sTime);

        /*计算执行上面的函数所需要的时间*/
gettimeofday(&tpstart,NULL);
function();
gettimeofday(&tpend,NULL);
timeuse=1000000*(tpend.tv_sec-tpstart.tv_sec)+
  tpend.tv_usec-tpstart.tv_usec;
timeuse/=100000;
printf("Used Time:%f\n",timeuse);
exit(0);
    }
```



8.3.2 用户任务中的定时器

在嵌入式系统中，定时器有着重要的作用，它可以在规定的时间到来时发出定时信号，中断当前任务运行，转而去处理定时信号。

在嵌入式 Linux 系统中，每个任务都自带有 3 个内部定时器可以用。

- ITIMER_REAL：这是一个实时定时器，用于计算减少的实际时间，定时到的时候发出 SIGALRM 信号。
- ITIMER_VIRTUAL：这不是一个实时时钟，它只计算任务占用 CPU 的时间（任务执行的时间）。任务执行时间达到规定时间后，将产生 SIGVTALRM 信号。
- ITIMER_PROF：计算任务占用 CPU 的有效时间和系统时间（为任务调度用的时间）。这个和上面一个的区别在于它包括系统内核调度时间和用户时间，它产生 SIGPROF 信号。

在任务中任一时刻每类定时器只能设置一个，在一个定时器的定时还没来到之前，又设置一个新的同类的定时器，则当前定时器将失效。

在任务中设置定时器的系统调用有 setitimer()、getitimer()或者 alarm()，具体函数定义如下：

```
#include <sys/time.h>
int setitimer(int whichtimer,struct itimerval *newval, struct itimerval *oldval);
int getitimer(int whichtimer,struct itimerval *value);
```

这其中牵涉到一个新的数据结构 itimeval，它的定义为：

```
struct itimerval {
struct timeval it_interval;
struct timeval it_value;
}
```

其中，it_interval 为连续的定时中断的时间间隔，如果它的值为 0，就表示只定时一次，it_value 为距离发出定时信号的时间，当这个值为 0 的时候就发出相应的信号，这时若 it_interval 不为 0，it_value 就会使自己的值等于 it_interval，再次设置好定时器。

getitimer()函数得到间隔计时器的时间值，并保存在 value 中。setitimer()函数设置间隔计时器的时间值为 newval，并将旧值保存在 oldval 中。which 表示使用 3 个计时器中的哪一个。

另外一个定时函数为 unsigned int alarm(unsigned int second)，它也是实现实时定时功能，但它的定时精度只能达到秒级。在定时的秒数到了之后，它也是发送 ITIMER_REAL

信号。事实上，alarm()函数可以用以下代码实现：

```
unsigned int alarm (unsigned int seconds)
{
struct itimerval old, new;
new.it_interval.tv_usec = 0;
new.it_interval.tv_sec = 0;
new.it_value.tv_usec = 0;
new.it_value.tv_sec = (long int) seconds;
if (setitimer (ITIMER_REAL, &new, &old) < 0)
return 0;
else
return old.it_value.tv_sec;
}
```

如果在程序中只是希望在原地等待一段时间，就可以调用 sleep()函数，它的调用形式为：unsinged int sleep(unsigned int second)，其精度为秒级。如果已经过了定时的时间，函数返回的是 0，如果由于一个信号而返回，则它的返回值为剩余的定时时间。

对于定时器信号，应该在程序中对之进行相应的处理。即在调用 settimer()或者 alarm()设置定时器之前，定义好信号处理函数。

以下是一个使用定时器的应用实例，在其中用到了定时器信号，每隔 3 秒钟就发出一次定时信号，在定时信号的处理函数中打印一次提示信息，表示 3 秒钟已经过去。具体实现代码如下（见光盘/demo/chap08/8-3）：

```
    /*定时器应用实例*/
    #include  <stdio.h>
#include  <signal.h>
#include  <time.h>


#define  PROMPT "时间已经过去了 3 秒钟\n\a"

char *prompt=PROMPT;
unsigned int len;

     /*信号处理函数*/
void prompt_info(int signo)
{
 write(STDERR_FILENO,prompt,len);
}
```




```
    /*初始化信号*/
void init_sigaction(void)
{
 struct sigaction act;
 act.sa_handler=prompt_info;
 act.sa_flags=0;
 sigemptyset(&act.sa_mask);
 sigaction(SIGPROF,&act,NULL);
}

    /*初始化代码*/
void init_time()
{
 struct itimerval value;
    /*定时 3 秒钟*/
 value.it_value.tv_sec=3;
 value.it_value.tv_usec=0;
 value.it_interval=value.it_value;
 setitimer(ITIMER_PROF,&value,NULL);
}

int main()
{
 len=strlen(prompt);
 init_sigaction();
 init_time();
 while(1);
 exit(0);
}
```

8.3.3 内核中的时钟管理

在 8.3.2 小节介绍了用户级的时钟管理，它需要得到 uClib 的支持，但是在编写设备驱动程序时，系统在内核中运行，这时不能使用 8.3.2 小节中的系统调用，而需要使用内核的时钟管理函数。

1. 时间片

在讨论内核的时钟管理时，首先要考虑的是系统的时间片（timeslice），也就是内核中产生时钟中断的时间间隔。

在嵌入式 Linux 操作系统中最重要的中断就是时钟中断，操作系统通过时钟中断来确定时间间隔。时钟中断的发生频率设定为 Hz，Hz 是一个与体系结构无关的常数，在文件<asm/param.h>中定义。例如，在<asm-i386/param-.h>中就有规定时钟中断发生频率的定义：

```
#ifdef _KERNEL_
# define CLOCKS_PER_SEC    100  /* frequency at which times() counts */
#endif
```

记录时钟中断发生的次数有一个重要的数据结构 jiffies。每当时钟中断发生时，jiffies 值就加 1。因此，jiffies 值就是自操作系统启动以来的时钟滴答的数目。jiffies 在头文件<linux/sched.h>中被定义的数据类型为：extern unsigned long volatile jiffies，它是 32 位无符号长整型的变量，因此连续累加一年又四个多月后就会溢出（假定 Hz=100，1 个 jiffies 等于 1/100 秒，jiffies 可记录的最大秒数为 42949672.96 秒，约合 1.38 年）。

如果修改 Hz 值后重新编译内核，在用户空间可能不会发现有什么不同，尽管 jiffies 值以不同的步长增长，但用户是感觉不到的。但如果修改幅度过大，则会产生一些问题，例如增加 Hz 的值，会产生更多的中断，系统开销更大了，但是因为处理器调度得更频繁了，系统会很不稳定。

2. 获取当前时间

内核一般通过 jiffies 值来获取当前时间。尽管该数值表示的是自上次系统启动到当前时间的时间间隔，但因为驱动程序的生命期只限于系统启动过后的时间（uptime），所以也是可行的。驱动程序可以利用 jiffies 的当前值来计算不同事件间的时间间隔。

其实驱动程序一般不需要知道墙上时间，需要墙上时间的情形是使用设备驱动程序的特殊情况。如果驱动程序真的需要获取当前时间，可以使用 do_gettimeofday()函数。该函数并不返回今天是本周的星期几或类似的信息，它是用微秒值来填充一个指向 struct timeval 的指针变量，其函数原型如下：

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
```

3. 使用定时器

在嵌入式 Linux 中使用定时器有长定时和短定时之分。长定时是指以时钟滴答为单位的定时，它可以在定时到了后进行规定的处理，类似于前面提到的 setitimer()。短定时是以微秒为单位的定时，它只是在原地等待规定的微秒。下面分别对它们进行说明。

（1）长定时

使用长定时，有一个重要的数据结构 struct timer_list，它的定义为：

```
struct timer_list {
```




```
struct timer_list *next;   /*不要直接修改它 */
truct timer_list *prev; /*不要直接修改它 */
unsigned long expires; /*timeout 超时值，以 jiffies 值为单位*/
unsigned long data; /*传递给定时器处理程序的参数*/
void (*function)(unsigned long); /*超时时调用的定时器处理程序*/
};
```

在使用时钟之前，需要首先声明一个 timer_list 结构，调用 init_timer 对它进行初始化。time_list 结构里 expires 是标明这个时钟的周期，单位采用 jiffies 的单位。jiffies 是 Linux 中一个全局变量，代表时间，它的单位随硬件平台的不同而有所不同。系统里定义了一个常数 Hz，代表每秒种最小时间间隔的数目，这样 jiffies 的单位就是 1/Hz，如 Intel 平台的 jiffies 的单位是 1/100 秒，这就是系统所能分辨的最小时间间隔了。所以 expires/Hz 就是以秒为单位的这个时钟的周期。当系统计时到达预定时间时就调用 function，并把此子程序从定时队列里删除，因此，如果想要每隔一定时间间隔执行一次的话，就必须在 function 里再一次调用 add_timer()。function 的参数 d 即为 timer_list 里面的 data 项。

在驱动程序中使用长定时的函数调用为：

```
#include <asm/param.h>
#include<linux/timer.h>
void add_timer(struct timer_list * timer);
int   del_timer(struct timer_list * timer);
inline void init_timer(struct timer_list * timer);
```

（2）短定时

如果要实现微秒级的定时，可以使用如下调用：

```
#include<asm/delay.h>
void udelay(unsigned long usecs);
```

udelay()函数在绝大多数体系结构上是作为内联函数编译的，并且使用软件循环将执行延迟指定数量的微秒数。这里要用到 BogoMips 值，udelay 利用了整数值 loops_per_second，这个值是在启动计算 BogoMips 时得到的。

udelay()函数只能用于获取较短的时间延迟，因为 loops_per_second 值的精度只有 8 位，所以当计算更长的延迟时会积累下相当大的误差。尽管运行的最大延迟将近 1 秒（因为更长的延迟将溢出），推荐的 udelay()函数的参数的最大值是取 1000 微秒（1 毫秒）。要特别注意的是 udelay()是个忙等待函数，在延迟的时间段内无法运行其他的任务。源码见头文件 <asm/delay.h>。

目前嵌入式 Linux 内核不支持大于 1 微秒而小于 1 个时钟滴答的延迟，但这不是个问题，因为延迟是给硬件或者人去识别的。百分之一秒的时间间隔对人来说其精度足够了，

而 1 毫秒对硬件来说延迟时间也足够长。如果用户真的需要这两者之间的延迟间隔，只要建立一个连续执行 udelay(1000)函数的循环。

8.4 小结

作为面向特定应用的嵌入式系统，提供人机接口常常是必要的，而且可以在最小系统上方便地扩展应用，这也正是嵌入式 Linux 的一大优势。本章讲述了如何把最简单的按键开关、矩阵键盘、标准键盘接入嵌入式 Linux 应用系统。其中包括与微处理器相连的方式，如何去除按键抖动，矩阵键盘扫描原理以及如何编写键盘驱动程序等，以使读者清晰地了解在嵌入式 Linux 下如何扩展自己的键盘应用。

同时，本章还介绍了有关时间定时器管理的相关知识。通过本章的学习，读者应当了解在嵌入式 Linux 下如何获取系统的时间；如何将系统时间转换为自己想要的格式；以及如何使用定时器等。

8.5 思考题

1. 试讲述按键开关的原理。
2. 按键抖动是如何产生的？如何去除按键抖动？
3. 行扫描法的原理是什么？它是如何得到按键扫描码的？
4. 请熟悉 Intel 8279 键盘/显示芯片，并了解如何在嵌入式系统使用 8279 扩展键盘。
5. 请编写一段代码，得到系统的日历时间，再得出今天是全年的第几天，并分别用字符串打印出来。
6. 系统提供的每个任务内的定时器有几个？它们的功能有什么不同？各用于什么场合？
7. 编写一段练习代码，代码中运行两个任务，两个任务用定时器进行阻塞，一个定时 2 秒，一个定时 3 秒，并且交替输出，试看输出效果。
8. 如果要在 x86 结构的系统内核中实现定时 2 秒，该如何实现？

第 9 章 图形界面应用程序开发

知识点：

- MiniGUI 的安装与配置
- MiniGUI 程序框架结构
- MiniGUI 图形界面编程实践

本章导读：

本章介绍嵌入式 Linux 环境下的 MiniGUI 图形界面应用程序开发。MiniGUI 是基于自由软件项目开发的一个轻量级的图形用户界面支持系统，它为在资源紧缺的嵌入式系统中实现图形界面显示提供良好支持。它的编程风格与在 Windows 环境下用 API 进行图形界面应用程序开发非常相似。本章首先介绍 MiniGUI 的安装与配置，然后以循序渐进的方式分析 MiniGUI 程序的结构及各图形界面元素的编程方法。

9.1 嵌入式 GUI 特点及种类

随着嵌入式系统的广泛应用，PDA、机顶盒、DVD/VCD 播放机及 WAP 手机已经迅速普及，而这些设备也同时被要求拥有华丽美观、易于操作的图形用户界面。

由于嵌入式系统实时性要求非常高，对 GUI 的要求也更高。这些系统一般不希望建立在庞大累赘的、非常消耗系统资源的操作系统和 GUI 之上，比如 Windows 或 X Window，这样，这些系统对轻型 GUI 的需求更加突出。另外，嵌入式系统往往是一种定制设备，对 GUI 的需求各不相同，有些系统只要求一些图形功能，而有些系统要求完备的 GUI 支持，因此，GUI 也必须是可定制的。嵌入式系统对 GUI 的基本要求包括轻型、占用资源少、高性能、高可靠性及可配置。

尽管实时嵌入式系统对 GUI 的需求越来越明显，但目前 GUI 的实现方法各有不同，主要有以下几点：

- 某些大型厂商有能力自己开发满足自身需要的 GUI 系统。
- 某些厂商没有将 GUI 作为一个软件层从应用程序中剥离，GUI 的支持逻辑由应用程序自己来负责。
- 采用某些比较成熟的 GUI 系统，比如 MiniGUI、MicroWindows 或者其他 GUI 系统。

比较常用的有如下几种 GUI 系统：精简的 X Window 系统、MiniGUI、MicroWindows、OpenGUI 及 QT/Embedded 等。下面对常用的系统介绍如下。

（1）MiniGUI

MiniGUI 由原清华大学教师魏永明先生开发，是一种面向嵌入式系统或者实时系统的图形用户界面支持系统。它主要运行于 Linux 控制台，实际可以运行在任何一种具有 POSIX 线程支持的 POSIX 兼容系统上。MiniGUI 同时也是国内最早出现的几个自由软件项目之一，本章 9.2 节将对它进行详细的介绍。

（2）MicroWindows

MicroWindows 是一个著名的开放源码的嵌入式 GUI 软件。MicroWindows 提供了现代图形窗口系统的一些特性。MicroWindows API 接口支持类 Win32 API，接口试图和 Win32 完全兼容，且还实现了一些 Win32 用户模块功能。MicroWindows 采用分层设计方法，以便不同的层面能够在需要时改写，基本上用 C 语言实现。MicroWindows 已经支持 Intel 16 位和 32 位 CPU、MIPS R4000 以及 ARM 芯片；但作为一个窗口系统，该项目提供的窗口处理功能还需要进一步完善，比如控件或构件的实现还很不完备，键盘和鼠标等的驱动还很不完善。

（3）OpenGUI

OpenGUI 在 Linux 系统上已经存在很长时间了。这个库是用 C++编写的，只提供 C++接口。OpenGUI 基于一个用汇编实现的 x86 图形内核，提供了一个高层的 C/C++图形/窗口接口。OpenGUI 提供了二维绘图函数原型、消息驱动的 API 及 BMP 文件格式支持。OpenGUI 功能强大，使用方便，支持鼠标和键盘的事件，在 Linux 上基于 Frame buffer 或者 SVGALib



实现绘图。由于其基于汇编实现的内核并利用 MMX 指令进行了优化，OpenGUI 运行速度非常快。但也正是由于其内核用汇编实现，可移植性受到了影响。

（4）QT/Embedded

QT/Embedded 是著名的 QT 库开发商 Trolltech 推出的面向嵌入式系统的 QT 版本。这个版本的主要特点是可移植性较好，许多基于 QT 的 X Window 程序可以非常方便地移植到嵌入式系统；但是该系统不是开放源码的，如果使用这个库，可能需要支付昂贵的授权费用。

9.2 MiniGUI 简介

9.2.1 MiniGUI 是什么

MiniGUI 是由北京飞漫软件技术有限公司主持的一个自由软件项目（遵循 LGPL 条款发布），其目标是为基于 Linux 的实时嵌入式系统提供一个轻量级的图形用户界面支持系统。

MiniGUI 为应用程序定义了一组轻量级的窗口和图形设备接口。利用这些接口，每个应用程序可以建立多个窗口，而且可以在这些窗口中绘制图形，用户也可以利用 MiniGUI 建立菜单、按钮、列表框等常见的 GUI 元素。

用户可以将 MiniGUI 配置成"MiniGUI-Threads"或者"MiniGUI-Lite"。运行在 MiniGUI-Threads 上的程序可以在不同的线程中建立多个窗口，但所有的窗口在一个进程中运行。相反，运行在 MiniGUI-Lite 上的每个程序是单独的进程，每个进程也可以建立多个窗口。MiniGUI-Threads 适合于具有单一功能的实时系统，而 MiniGUI-Lite 则适合于类似于 PDA 和瘦客户机等复杂嵌入式系统。

MiniGUI 最新稳定版是 1.3.0。

9.2.2 MiniGUI 特点及优势

1. 特点

MiniGUI 具有以下特点：

- 是遵循 LGPL 条款的纯自由软件。
- 提供了完备的多窗口机制。
- 提供对话框和预定义的控件类（按钮、单行和多行编辑框、列表框、进度条、工具栏等）。
- 消息传递机制。
- 多字符集和多字体支持，目前支持 ISO8859-1、GB2312、Big5 等字符集，并且支持各种光栅字体和 TrueType、Type 1 等矢量字体。
- 支持全拼、五笔等汉字输入法。
- 支持 BMP、GIF、JPEG、PCX、TGA 等常见的图像文件。

- Windows 的资源文件支持，如位图、图标、光标等。
- 插入符、定时器、加速键等。

2. 优势

与其他嵌入式图形界面支持系统相比，MiniGUI 具有以下优势：

- 小巧：包含全部功能的库文件大小为 300 KB 左右。
- 可配置：可根据项目需求进行定制配置和编译。
- 高稳定性和高性能：MiniGUI 已经在 Linux 发行版安装程序、CNC 系统、蓝点嵌入式系统等关键应用程序中得到了实际的应用。
- 可移植性好：目前，MiniGUI 可以在 X Window 和 Linux 控制台上运行。中科院 EEOS 开发组已经成功地将 MiniGUI 移植到了 POSIX 兼容系统上。蓝点软件（北京）研发中心也已经成功地将 MiniGUI 移植到了两款基于 StrongARM 的嵌入式系统上。

9.2.3 MiniGUI 的安装与配置

为了方便读者理解，这里简要介绍 MiniGUI 在 PC 机上的安装及配置，PC 机上 Linux 内核版本为 2.4.18，MiniGUI 版本为 1.2.3。

1. 编译并安装 MiniGUI

以 ROOT 权限登录 Linux，编辑/boot/grub/grub.conf 文件，在其中加入以下几行系统引导选项设置命令：

```
title MiniGUI(1.2.3)
    root(hd0,7)
    kernel /vmlinuz-2.4.18-3 ro root=/dev/hda9 vga=0x0317 fb:on
    initrd /initrd-2.4.18-3.img
```

这个文件是 grub 引导程序配置文件，其中最重要的是第 3 行后面加上的 vga=0x0317 fb:on，表示设置显示器分辨率为 1024×768，颜色位数为 16 位，并打开内核的 FrameBuffer 显示模式。当然，如果引导程序是 lilo，则应依据 lilo 配置文件格式进行相应设置。

加上这几行引导选项设置命令后，重新启动计算机，这时系统引导界面的引导列表中多了一项：MiniGUI(1.2.3)，选择该项进入 Linux，则内核为 MiniGUI 正常安装及运行做好了准备。

关于 VGA 值与显示器分辨率的关系如表 9-1 所示。

表 9-1 显示模式对照表

	640×480	800×600	1024×768	1280×1024
8 位色	0x301	0x303	0x305	0x307
15 位色	0x310	0x313	0x316	0x319
16 位色	0x311	0x314	0x317	0x31A
24 位色	0x312	0x315	0x318	0x31B



安装 MiniGUI 前，首先确保下载了以下压缩软件包：

- libminigui-1.2.3.tar.gz：MiniGUI 函数库源代码。
- minigui-res-1.2.0.tar.gz：MiniGUI 资源文件。
- minigui-fonts-1.1.0.tar.gz：MiniGUI 基本字体文件。
- minigui-imetabs-1.1.0.tar.gz：MiniGUI 中文 GB 输入法码表。
- mde-1.2.3.tar.gz：MiniGUI 示例程序源代码。

下面开始安装 MiniGUI 软件包，这些软件包都是使用 autoconf 和 automake 接口的，所以配置和安装都非常简单。MiniGUI 软件包可以到北京飞漫软件公司网站 www.minigui.com 下载。

（1）安装 MiniGUI 资源文件

用 tar 命令解开资源文件压缩包：

```
tar –zxvf minigui-res-1.2.0.tar.gz
```

完成后进入新建的目录 minigui-res 中，运行 make 命令：

```
make install
```

这样，资源文件便被正确安装了。

以同样的方式安装字体软件包 minigui-fonts-1.1.0.tar.gz 及输入法软件包 minigui-imetabs-1.1.0.tar.gz，这里不再赘述。

（2）装 MiniGUI 函数库

首先用 tar 命令解开函数库文件压缩包：

```
tar –zxvf libminigui-1.2.3.tar.gz
```

完成后进入新建的目录 libminigui-1.2.3，运行 autogen 命令：

```
./autogen.sh
```

完毕后再执行 configure 命令：

```
./configure
```

然后编译并安装函数库：

```
make;
make install
```

还要在 Linux 的系统共享函数库配置文件/etc/ld.so.conf 中加入 MiniGUI 的函数库路径，加上下面这行路径设置选项：

```
/usr/local/lib
```

最后执行 ldconfig 命令更新系统函数库缓存。至此，MiniGUI 已经在系统中正确安装。

2. MiniGUI 演示程序的安装与运行

首先用 tar 命令解开演示例子源代码压缩包：

```
tar mde-1.2.3.tar.gz
```

完成后进入新建的目录 mde-1.2.3，运行 autogen 命令：

```
./autogen.sh
```

完成后再执行./configure 命令，最后执行 make 命令安装范例程序。

安装完成后，进入/usr/local/etc 目录，修改 MiniGUI 配置文件 MiniGUI.cfg，找到以下两处参数设置项：

```
[fbcon]
    defaultmode=800×600-16bpp
[qvfb]
    defaultmode=800×600-16bpp
    display=0
```

将 defaultmode 改为想使用的分辨率，如 1024×768-16bpp，保存文件后退出。

至此，演示程序安装完毕。下面可以看看劳动成果了，进入 mde-1.2.3/mginit 目录，执行./mginit，怎么样？MiniGUI 视窗界面很漂亮吧？

9.3 MiniGUI 程序框架及示例

在具体了解 MiniGUI 图形界面应用程序的编程细节之前，先来看看一个完整的 MiniGUI 程序应该是什么样子，有 Windows 环境下的编程经验的读者可以将它与 Windows SDK 程序作一下比较，可以看出二者在结构与概念上都非常相似。

9.3.1 主函数 MiniGUIMain()

MiniGUI 的程序入口点是 MiniGUIMain()函数，它负责创建程序的主窗口，类似 Windows 程序中的 WinMain()函数。MiniGUIMain()函数首先初始化一个 MAINWINCREATE 结构，该结构中的元素定义如下：

```
typedef struct _MAINWINCREATE
{
    DWORD dwStyle;              /*窗口风格*/
    DWORD dwExStyle;            /*窗口的附加风格*/
    const char* spCaption;      /*窗口的标题*/
    HMENU hMenu;                /*附加在窗口上的菜单句柄*/
    HCURSOR hCursor;            /*在窗口中所使用的鼠标光标句柄*/
    HICON hIcon;                /*程序的图标*/
    HWND   hHosting;            /*消息队列所属窗口*/
    /*窗口的消息处理函数指针 */
    int (*MainWindowProc)(HWND, int, WPARAM, LPARAM);
    int lx, ty, rx, by;          /*主窗口在屏幕上的位置及大小*/
    int iBkColor;               /*窗口背景颜色*/
    DWORD dwAddData;            /*附带给窗口的一个 32 位值*/
```




```
    DWORD dwReserved;        /*保留域*/
}MAINWINCREATE;
```

其中有几个域需要特别说明：

- dwAddData：在程序编制过程中，应该尽量减少静态变量，但是如何不使用静态变量而给窗口传递参数呢？这时可以使用这个域。该域是一个 32 位的值，因此可以把所有需要传递给窗口的参数编制成一个结构，而将结构的指针赋予该域。在窗口过程中，可以使用 GetWindowAdditionalData()函数获取该指针，从而获得所需要传递的参数。
- hHosting：该域表示的是将要建立的主窗口使用哪个主窗口的消息队列。使用其他主窗口消息队列的窗口称为“被托管”的主窗口。当然，这只在 MiniGUI-Threads 版本中有效。

MainWinProc()函数负责处理窗口消息。这个函数就是主窗口的“窗口过程”。窗口过程一般有 4 个入口参数，第一个是窗口句柄，第二个是消息类型，第三个和第四个是消息的两个参数。

在准备好 MAINWINCREATE 结构之后，就可以调用 CreateMainWindow()函数建立主窗口了。

在建立主窗口之后，程序进入消息循环。

9.3.2 消息处理函数

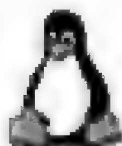
MiniGUI 的消息循环就是一个循环体，在这个循环体中，程序利用 GetMessage()函数不停地从消息队列中获得消息，然后利用 DispatchMessage()函数将消息发送到指定的窗口，并传递消息及其参数，窗口的消息处理函数根据接收到的不同消息分别进行处理。

9.3.3 第一个 MiniGUI 程序

一个简单的示例程序，该程序在窗口中打印“Hello, world!”，其源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <minigui/common.h>
#include <minigui/minigui.h>
#include <minigui/gdi.h>
#include <minigui/window.h>
/*消息处理函数*/
static int HelloWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
```

```
{
    HDC hdc;
    switch (message) {
        case MSG_PAINT:
            hdc = BeginPaint (hWnd);
            TexOut (hdc, 0, 0, "Hello, world!");
            EndPaint (hWnd, hdc);
        break;
        case MSG_CLOSE:
            DestroyMainWindow (hWnd);
            PostQuitMessage (hWnd);
        return 0;
    }
    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}
/*窗口参数结构初始化*/
static void InitCreateInfo (PMAINWINCREATE pCreateInfo)
{
    CreateInfo.dwStyle = WS_VISIBLE | WS_VSCROLL |
                         WS_HSCROLL | WS_CAPTION;
    CreateInfo.spCaption= "MiniGUI step three";
    CreateInfo.dwExStyle = WS_EX_NONE;
    CreateInfo.hMenu = createmenu();
    CreateInfo.hCursor = GetSystemCursor(0);
    CreateInfo.hIcon = 0;
    CreateInfo.MainWindowProc = MainWinProc;
    CreateInfo.lx = 0;
    CreateInfo.ty = 0;
    CreateInfo.rx = 640;
    CreateInfo.by = 480;
    CreateInfo.iBkColor = COLOR_lightwhite;
    CreateInfo.dwAddData = 0;
    CreateInfo.hHosting = HWND_DESKTOP;
}
/*主函数*/
int MiniGUIMain (int args, const char* arg[])
{
    MSG Msg;
    MAINWINCREATE CreateInfo;
    HWND hWnd;
     /*初始化 MAINWINCREATE 结构*/
    InitCreateInfo (&CreateInfo);
    /*建立主窗口*/
```




```
    hWnd = CreateMainWindow(&CreateInfo);
    if (hWnd == HWND_INVALID)
        return 0;
    /*显示主窗口*/
    ShowWindow (hWnd, SW_SHOWNORMAL);
    /*进入消息循环*/
    while (GetMessage(&Msg, hWnd)) {
        TranslateMessage (&Msg);
        DispatchMessage(&Msg);
    }
    MainWindowThreadCleanup (hWnd);
    return 0;
}
```

很显然，这是个非常简单的程序。该程序使用了 MiniGUI 的默认过程来处理前面提到的许多消息，而仅仅处理了 MSG_PAINT 和 MSG_CLOSE 两条消息。当用户单击标题栏上的关闭按钮时，MiniGUI 将发送 MSG_CLOSE 到窗口过程，这时应用程序就可以销毁窗口，并终止消息循环，最终退出程序。

9.4 MiniGUI 中的窗口与消息

9.4.1 窗口的建立与销毁

1. 窗口的建立

MiniGUI 的窗口的建立过程与 Windows 程序基本类似，不过也有一些差别。在 Windows 程序中，在建立一个主窗口之前，程序首先要注册一个窗口类，然后创建一个属于该窗口类的主窗口。MiniGUI 却没有在主窗口中使用窗口类的概念。在 MiniGUI 程序中，调用 CreateMainWindow()函数建立主窗口，建立主窗口之后，程序将进入消息循环。

和 Windows 程序不同的是在退出消息循环之后，还要调用 MainWindowThreadCleaup()函数以销毁主窗口的消息队列，该函数一般在线程或者进程的最后调用。

2. 窗口的销毁

要销毁一个主窗口，可以利用 DestroyMainWindow()函数。该函数可以销毁主窗口，但不会销毁主窗口所使用的消息队列，而要使用 MainWindowThreadCleaup()函数最终清除主窗口所使用的消息队列。

一般而言，一个主窗口过程在接收到 MSG_CLOSE 消息之后会销毁主窗口，并调用 PostQuitMessage 消息终止消息循环。销毁窗口的函数代码如下：

```
case MSG_CLOSE:
/*销毁窗口使用的资源*/
DestroyLogFont (logfont1);
DestroyLogFont (logfont2);
DestroyLogFont (logfont3);
/*销毁子窗口*/
DestroyWindow(hWndButton);
DestroyWindow(hWndEdit);
/*销毁主窗口*/
DestroyMainWindow (hWnd);
/*发送 MSG_QUIT 消息*/
PostQuitMessage(hWnd);
return 0;
```

9.4.2 消息与消息循环

1. 消息

在 MiniGUI 中，消息定义的源代码（resource/minigui1.3.0/src/include/window.h）如下：

```
typedef struct _MSG
{
    HWND     hwnd;
    int      message;
    WPARAM   wParam;
    LPARAM   lParam;
    #ifdef _LITE_VERSION
    unsigned int    time;
    #else
    struct timeval    time;
    #endif
    POINT    pt;
    #ifndef _LITE_VERSION
    void*    pAdd;
    #endif
}MSG;
```

一个消息由该消息所属的窗口（hwnd）、消息编号（message）、消息的 WPARAM 型参数（wParam）以及消息的 LPARAM 型参数（lParam）组成。消息的两个参数中包含了重要的内容。比如，对鼠标消息而言，lParam 中一般包含鼠标的位置信息，而 wParam 参数



中则包含发生该消息时，对应的 Shift 键的状态信息等。对其他不同的消息类型来讲，wParam 和 lParam 也具有明确的定义。当然，用户也可以自定义消息，并定义消息的 wParam 和 lParam 意义。为了使用户能够自定义消息，MiniGUI 定义了 MSG_USER 宏，可如下定义自己的消息：

```
#define MSG_MYMESSAGE1    (MSG_USER + 1)
#define MSG_MYMESSAGE2    (MSG_USER + 2)
```

用户可以在自己的程序中使用自定义消息，并利用自定义消息传递数据。

2. 消息循环

消息循环就是一个循环体，在这个循环体中，程序利用 GetMessage()函数不停地从消息队列中获得消息，然后利用 DispatchMessage()函数将消息发送到指定的窗口，也就是调用指定窗口的消息处理函数，并传递消息及其参数。典型的消息循环代码示例如下：

```
while (GetMessage (&Msg, hMainWnd)) {
        TranslateMessage (&Msg);
        DispatchMessage (&Msg);
}
```

在 MiniGUI-Threads 版本中，每个建立有窗口的 GUI 线程有自己的消息队列，而且，所有属于同一线程的窗口共享同一个消息队列。因此，GetMessage()函数将获得所有与 hMainWnd 窗口在同一线程中的窗口的消息。

而在 MiniGUI-Lite 版本中，只有一个消息队列，GetMessage()函数将从该消息队列中获得所有的消息，而忽略 hMainWnd 参数。

除了上面提到的 GetMessage()函数、TranslateMessage()函数和 DispatchMessage()函数以外，MiniGUI 还支持如下几个消息处理函数。

（1）PostMessage()函数

该函数将消息放到指定窗口的消息队列后立即返回，这种发送方式称为"邮寄"消息。如果消息队列中的邮寄消息缓冲区已满，则该函数返回错误值。在下一个消息循环中，由 GetMessage()函数获得这个消息之后，窗口才会处理该消息。PostMessage()函数一般用于发送一些非关键性的消息。比如在 MiniGUI 中，鼠标和键盘消息就是通过 PostMessage()函数发送的。

（2）SendMessage()函数

SendMessage()函数和 PostMessage()函数不同，它在发送一条消息给指定窗口时，将等待该消息被处理之后才会返回。当需要知道某个消息的处理结果时，使用该函数发送消息，然后根据其返回值进行处理。在 MiniGUI-Threads 中，如果发送消息的线程和接收消息的线程不是同一个线程，发送消息的线程将阻塞并等待另一个线程的处理结果，然后继续运

行；否则，SendMessage()函数将直接调用接收消息窗口的窗口过程函数。而 MiniGUI-Lite 则直接调用接收消息窗口的窗口过程函数。

（3）SendNotifyMessage()函数

该函数和 PostMessage 消息类似，也是不等待消息被处理即返回。但和 PostMessage 消息不同，通过该函数发送的消息不会因为缓冲区满而丢失。通过该函数发送的消息一般称为"通知消息"，一般用来从控件向其父窗口发送通知消息。

（4）PostQuitMessage()函数

该消息在消息队列中设置一个 QS_QUIT 标志，GetMessage 在从指定消息队列中获取消息时，会检查该标志，如果有 QS_QUIT 标志，GetMessage 消息将返回 FALSE，从而可以利用该返回值终止消息循环。

9.4.3 几个重要的消息

在窗口（包括主窗口和子窗口在内）的生存周期中，有几个重要的消息需要仔细处理。

1. MSG_NCCREATE 消息

该消息在 MiniGUI 建立主窗口的过程中发送到窗口过程。lParam 中包含了由 CreateMainWindow 传递进入的 pCreateInfo 结构指针。可以在该消息的处理过程中修改 pCreateInfo 结构中的某些值，例如主窗口标题属性值 spCaption、主窗口图标属性值 hMenu 等。

2. MSG_SIZECHANGING 消息

该消息在建立窗口或窗口尺寸发生变化时发送到窗口过程，用来确定窗口大小。wParam 包含预期的窗口尺寸值，而 lParam 用来保存结果值。MiniGUI 的默认处理代码如下：

```
case MSG_SIZECHANGING:
    memcpy ((PRECT)lParam, (PRECT)wParam, sizeof (RECT));
    return 0;
```

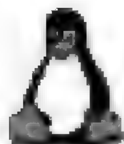
可以截获该消息的处理，从而让即将创建的窗口位于指定的位置，或者具有固定的大小。

3. MSG_CHANGESIZE 消息

在确定窗口大小之后，该消息被发送到窗口过程，用来通知确定之后的窗口大小。wParam 包含了窗口大小 RECT 的指针，应用程序应该将该消息传递给 MiniGUI 进行默认处理。

4. MSG_SIZECHANGED 消息

该消息用来确定窗口客户区的大小和 MSG_SIZECHANGING 消息类似，wParam 参数



包含窗口大小信息；lParam 参数是用来保存窗口客户区大小的 RECT 指针，并且具有默认值。如果该消息的处理返回非零值，则将采用 lParam 当中包含的大小值作为客户区的大小；否则，将忽略该消息的处理。

5. MSG_CREATE 消息

该消息在建立好的窗口成功添加到 MiniGUI 的窗口管理器之后发送到窗口过程。这时，应用程序可以在其中创建子窗口。如果该消息返回非零值，则将销毁新建的窗口。

注意：在 MSG_NCCREATE 消息被发送时，窗口尚未正常建立，所以不能在 MSG_NCCREATE 消息中建立子窗口。

6. MSG_PAINT 消息

该消息在需要进行窗口重绘时发送到窗口过程。MiniGUI 通过判断窗口是否含有无效区域来确定是否需要重绘。当窗口在初始显示、从隐藏状态变化为显示状态、从部分不可见到可见状态，或者应用程序调用 InvalidateRect()函数使某个矩形区域变成无效时，窗口将具有特定的无效区域。这时，MiniGUI 将在处理完所有的邮寄消息、通知消息之后处理无效区域，并向窗口过程发送 MSG_PAINT 消息。该消息的典型处理代码如下：

```
case MSG_PAINT:
{
    HDC hdc;
    hdc = BeginPaint (hWnd);
    /*使用 hdc 绘制窗口*/
    ...
    EndPaint (hWnd, hdc);
    break;
}
```

7. MSG_DESTROY 消息

该消息在应用程序调用 DestroyMainWindow()或者 DestroyWindow)()时发送到窗口过程中，用来通知系统即将销毁一个窗口。如果该消息的处理返回非零值，则取消销毁过程。

9.5 键盘与鼠标

键盘与鼠标作为输入设备，是用户与应用程序进行交互的重要手段。键盘上的每个键对应一个惟一的键值，称为扫描码。当用户按下一个键或组合键时，将产生一条消息，系统将该消息送入对应程序的消息队列中，应用程序再通过消息循环中的 GetMessage()函数从应用程序当前窗口所属的消息队列中获得该消息，最后调用 DispatchMessage()函数将消息发送到指定的窗口过程函数中进行处理。鼠标的工作方式与键盘相似，MiniGUI 对鼠标

消息的处理方式也与对键盘消息的处理方式相似。

9.5.1 键盘消息与字符消息

当用户按下一个键或组合键时，将产生 MSG_KEYDOWN 或 MSG_SYSKEYDOWN 键盘消息；当按键被释放时将产生 MSG_KEYUP 或 MSG_SYSKEYUP 键盘消息。这些消息的 wParam 参数保存按键的扫描码，lParam 参数中保存该消息的附加信息，如是否同时按下了 Ctrl 键等。应用程序对键盘消息的处理方式示例代码如下：

```
switch (message) {
    case MSG_KEYDOWN:
        if (wParam ==SCANCODE_C && lParam & KS_CTRL) {
            /*用户按下 Ctrl+C 组合键*/
            break;
        }
        break;
    ...
}
```

提示：MSG_SYSKEYDOWN 或 MSG_SYSKEYUP 为系统键消息，由输入键与 Alt 键组合产生。

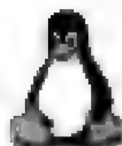
当按键能产生可见字符时，则除了产生键盘消息外还将产生字符消息。消息循环中的 TranslateMessage()函数负责判断按键是否产生了可见字符，如果是可见字符，TranslateMessage()函数将键盘消息转化为字符消息，以便应用程序进行相应处理。字符消息为 MSG_CHAR 或 MSG_SYSCHAR。与键盘消息不同的是，字符消息的 wParam 参数中保存的是按键的 ASCII 码，lParam 参数中保存的附加信息为按键是否同时按下了 Shift 键。

9.5.2 鼠标消息

鼠标的动作方式比键盘丰富得多，它可以移动、单击、双击，所处位置可以在客户区，也可以在非客户区。所以，鼠标消息的种类也比键盘消息多。MiniGUI 定义了 7 个客户区鼠标消息和 7 个非客户区鼠标消息，一般来说，应用程序只处理客户区鼠标消息。客户区鼠标消息如表 9-2 所示。

表 9-2 客户区鼠标消息

消息名称	鼠标操作
MSG_LBUTTONDOWN	按下鼠标左键
MSG_LBUTTONUP	松开鼠标左键



（续）

消 息 名 称	鼠 标 操 作
MSG_LBUTTONDBLCLK	双击鼠标左键
MSG_MOUSEMOVE	鼠标移动
MSG_RBUTTONDOWN	按下鼠标右键
MSG_RBUTTONUP	松开鼠标右键
MSG_RBUTTONDBLCLK	双击鼠标右键

鼠标消息的 wParam 参数保存附加信息，判断击键时键盘的 Shift 键是否被同时按下；lParam 参数保存鼠标在当前客户区的位置坐标。

应用程序对鼠标消息的处理示例如下：

```
switch (message) {
    case MSG_LBUTTONDOWN:
        if (wParam & KS_RIGHTBUTTON) {
            /*鼠标左键与右键被同时按下*/
            break;
        }
        break;
    case MSG_RBUTTONDOWN:
        if (wParam & KS_LEFTBUTTON) {
            /*鼠标左键与右键被同时按下*/
            break;
        }
        break;
    case MSG_MOUSEMOVE:
        if (wParam & KS_CTRL) {
            /*按下 Ctrl 键并拖动鼠标*/
            break;
        }
        break;
    ...
}
```

9.6 绘图工具与图形设备接口

GUI 系统的一个重要组成部分就是 GDI，即图形设备接口（Graphics Device Interface），通过 GDI，GUI 程序就可以在计算机屏幕上或者其他的显示设备上进行图形输出，包括基本绘图和文本输出。

9.6.1 设备描述表

MiniGUI 中的图形设备概念与 Windows 中的相同，每个图形设备定义了计算机显示屏幕上的一个矩形输出区域。在调用图形输出函数时，均要求指定经初始化的图形设备上下文（Device Context，DC），也称做“设备环境”。从程序员的角度看，一个经过初始化的图形设备上下文确定了其后进行图形输出的一些基本属性，并一直保持这些属性，直到被改变为止。这些属性包括：输出的线条颜色、填充颜色、字体颜色、字体形状等。

1. 设备上下文的获取和释放

在 MiniGUI 中，所有绘图相关的函数均需要有一个设备上下文。设备上下文可通过 GetDC()函数、GetClientDC()函数和 ReleaseDC()函数获取和释放。由 GetDC()函数所获取的设备上下文是针对整个窗口的，而 GetClientDC()函数所获取的设备上下文是针对窗口客户区的；GetDC()函数获得的设备上下文，其坐标原点位于窗口左上角，输出被限定在窗口范围之内；GetClientDC()函数获得的设备上下文，其坐标原点位于窗口客户区左上角，输出被限定在窗口客户区范围之内。下面是这 3 个函数的定义说明(resource/minigui1.3.0/src/include/gdi.h)：

```
HDC GUIAPI GetDC (HWND hwnd);
HDC GUIAPI GetClientDC (HWND hwnd);
void GUIAPI ReleaseDC (HDC hdc);
```

GetDC()函数和 GetClientDC()函数是从系统预留的若干个 DC 当中获得一个目前尚未使用的设备上下文。所以，应该注意以下两点：

- 在使用完一个由 GetDC 返回的设备上下文之后，应该尽快调用 ReleaseDC 释放。
- 避免同时使用多个设备上下文，并避免在递归函数中调用 GetDC 和 GetClientDC。

为了方便程序编写，提高绘图效率，MiniGUI 还提供了建立私有设备上下文的函数，所建立的设备上下文在整个窗口生存期内有效，从而免除了获取和释放的过程。这些函数的定义如下：

```
HDC GUIAPI CreatePrivateDC (HWND hwnd);
HDC GUIAPI CreatePrivateClientDC (HWND hwnd);
HDC GUIAPI GetPrivateClientDC (HWND hwnd);
void GUIAPI DeletePrivateDC (HDC hdc);
```

在建立主窗口时，如果主窗口的扩展风格中指定了 WS_EX_USEPRIVATEDC 风格，则 CreateMainWindow()函数会自动为该窗口的客户区建立私有设备上下文。通过 GetPrivateClientDC()函数，可以获得该设备上下文。对控件而言，如果控件类具有



CS_OWNDC 属性，则所有属于该控件类的控件将自动建立私有设备上下文。DeletePrivateDC()函数用来删除私有设备上下文，系统将在销毁窗口时自动调用 DeletePrivateDC()函数。

另外一个获取和释放设备上下文的方法是通过 BeginPaint 和 EndPaint()函数。这两个函数只能在处理 MSG_PAINT 的消息中调用。这两个函数的定义如下（resource/minigui1.3.0/src/include/window.h）：

```
HDC GUIAPI BeginPaint(HWND hWnd);
void GUIAPI EndPaint(HWND hWnd, HDC hdc);
```

2. 内存设备上下文

MiniGUI 也提供了内存设备上下文的创建和销毁函数。利用内存设备上下文，可以在系统内存中建立一个类似显示内存的区域，然后在该区域中进行绘图操作，结束后再复制到显示内存中。这种绘图方法有许多好处，比如速度很快、减少直接操作显存造成的闪烁现象等。不过，目前 MiniGUI 中只能建立和显示内存，也就是与物理设备上下文一样的内存设备上下文。用来建立和销毁内存设备上下文的函数定义如下（resource/minigui1.3.0/src/include/gdi.h）：

```
HDC GUIAPI CreateCompatibleDC (HDC hdc);
void GUIAPI DeleteCompatibleDC (HDC hdc);
```

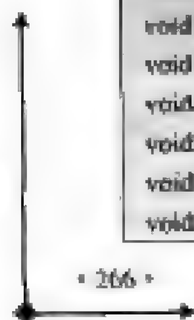
3. 屏幕设备上下文

MiniGUI 在启动之后，就建立了一个全局的屏幕设备上下文，屏幕设备上下文用 HDC_SCREEN 标识，不需要进行任何获取和释放操作。

4. 映射模式

一个设备上下文被初始化之后，其坐标系原点通常是输出矩形的左上角，而 X 轴水平向左，Y 轴垂直向下，并以像素为单位。这种坐标的映射模式标识为 MM_TEXT。MiniGUI 提供了一套函数，可以改变这种映射方式，包括对默认坐标系进行偏移、缩放等操作。这些函数的定义如下（resource/minigui1.3.0/src/include/gdi.h）：

```
int GUIAPI GetMapMode (HDC hdc);
void GUIAPI GetViewportExt (HDC hdc, POINT* pPt);
void GUIAPI GetViewportOrg (HDC hdc, POINT* pPt);
void GUIAPI GetWindowExt (HDC hdc, POINT* pPt);
void GUIAPI GetWindowOrg (HDC hdc, POINT* pPt);
void GUIAPI SetMapMode (HDC hdc, int mapmode);
void GUIAPI SetViewportExt (HDC hdc, POINT* pPt);
void GUIAPI SetViewportOrg (HDC hdc, POINT* pPt);
void GUIAPI SetWindowExt (HDC hdc, POINT* pPt);
void GUIAPI SetWindowOrg (HDC hdc, POINT* pPt);
```



GetMapMode()函数返回当前的映射模式，若不是 MM_TEXT 模式，则返回 MM_ANISOTROPIC。SetMapMode()函数设置映射模式，MiniGUI 目前只支持两种映射模式，即 MM_ANISOTROPIC 和 MM_TEXT。Get()函数组用来返回映射模式信息，包括偏移量、缩放比例等；而 Set()函数组用来设置相应的映射信息。

通常情况下，MiniGUI 的 GDI()函数所指定的坐标参数称为"逻辑坐标"，在绘制之前，首先要转化成"设备坐标"。当使用 MM_TEXT 映射模式时，逻辑坐标和设备坐标是等价的。LPtoDP()函数用来完成逻辑坐标到设备坐标的转换，DPtoLP()函数用来完成从设备坐标到逻辑坐标的转换。另外，LPtoSP()函数和 SPtoLP()函数用来完成逻辑坐标和屏幕坐标之间的转换。

9.6.2 画点与画线

1. 画点

画点是最基本的画图操作，MiniGUI 中通过 SetPixel()函数来实现。SetPixel()函数定义如下：

```
void GUIAPI SetPixel (HDC hdc, int x, int y, gal_pixel pixel);
```

其中参数 hdc 为设备环境句柄，x 和 y 为画点的坐标值，pixel 为画点的颜色值。

2. 画直线

画直线也是非常常见的画图操作，通过 LineTo()和 MoveTo()函数来实现。两个函数配合使用，就可以完成任何直线和折线的操作。LineTo()函数以当前点为起始点，通过指定直线终点画出一条直线。函数定义如下（resource/minigui1.3.0/src/include/gdi.h）：

```
void GUIAPI LineTo (HDC hdc, int x, int y);
```

其中参数 hdc 为设备环境句柄，x 和 y 为直线终点坐标值。需要说明的是，如果函数调用成功，则该直线终点值自动成为当前点。

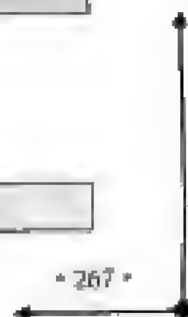
MoveTo()函数用来更新当前点坐标，函数定义如下：

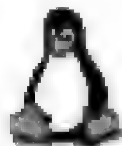
```
void GUIAPI MoveTo (HDC hdc, int x, int y);
```

3. 画折线与弧线

PolylineTo()函数用于画一条折线，函数定义如下：

```
void GUIAPI PolylineTo (HDC hdc, const POINT* pts, int vertices);
```





其中参数 hdc 为设备环境句柄，pts 是折线顶点数组的指针，vertices 参数指定折线顶点数组中的顶点数。

注意：由于一条折线至少有两个顶点，所以 vertices 参数的值不应小于 2。与 LineTo() 函数一样，PolylineTo()函数也是将当前位置作为起始点，并且在画完折线后，把折线终点所在位置设为新的当前位置。下面的示范程序段用来绘制一个五角星形：（demo/gdi.h）

```
const POINT   point[6]={{100,50},{71,141},{148,85},{52,85},{129,141},{100,50}};
PolylineTo(hdc,point,6);
```

弧线的绘制函数为 Arc()，函数定义如下：

```
void GUIAPI Arc (HDC hdc, int sx, int sy, int r, fixed ang1, fixed ang2);
```

其中参数 hdc 为设备环境句柄，sx 和 sy 为弧线中点坐标，r 为弧线半径，ang1 和 ang2 分别为弧线的起始点角度和终点角度。

9.6.3 封闭曲线及区域填充

1. 封闭曲线

MiniGUI 提供了一组画封闭曲线的函数，主要有 Rectangle()、Ellipse()、Circle()，分别用来画矩形、椭圆和圆。这些函数分别定义如下：

```
void GUIAPI Rectangle (HDC hdc, int x0, int y0, int x1, int y1);
void GUIAPI Ellipse (HDC hdc, int sx, int sy, int rx, int ry);
void GUIAPI Circle (HDC hdc, int sx, int sy, int r);
```

其中参数 hdc 均为设备环境句柄，其余参数如下：

- Rectangle()函数：x0、y0 为矩形左上角点坐标，x1、y1 为矩形右下角点坐标。
- Ellipse()函数：sx、sy 为椭圆中心点坐标，rx、ry 分别为椭圆长短轴半径。
- Circle()函数：sx、sy 为圆中心点坐标，r 为圆半径。

2. 区域填充

绘图时往往需要以某一颜色填充一个区域，MiniGUI 提供了丰富的区域填充函数，说明如下：

- FillBox()函数：填充一个矩形框。
- FillCircle()函数：填充一个圆。



- FillEllipse()函数：填充一个椭圆。
- FillPolygon()函数：填充一个多边形。
- FloodFill()函数：填充一个封闭区域。

这些函数的参数都与对应的封闭曲线绘制函数相同，这里不再详述。

9.6.4 字体与文字输出

1. 字体

在 MiniGUI 中，应用程序对字体的操作是通过逻辑字体来进行的，每个设备上下文的默认逻辑字体是系统字体，即用来显示菜单、标题的逻辑字体。逻辑字体可以通过调用 CreateLogFont()和 CreateLogFontIndirect()两个函数来建立，并利用 SelectFont()函数将逻辑字体选择到指定的设备上下文中，在使用结束之后，用 DestroyLogFont()函数销毁逻辑字体。这几个函数的原型如下（include/gdi.h）：

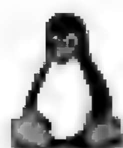
```
PLOGFONT GUIAPI CreateLogFont (const char* type, const char* family,
                    const char* charset, char weight, char slant, char set_width,
                    char spacing, char underline, char struckout,
                    int size, int rotation);
PLOGFONT GUIAPI CreateLogFontIndirect (LOGFONT* logfont);
PLOGFONT GUIAPI SelectFont (HDC hdc, PLOGFONT log_font);
void GUIAPI DestroyLogFont (PLOGFONT log_font);
```

其中，CreateLogFont()函数的参数说明如下：

- type：逻辑字体。
- family：字体系列名称，如宋体、楷体等。
- charset：字体所属字符集，如 ISO8859-1、GB2312.1980-0 等。
- weight：字体粗细，可以为常规或粗体。
- slant：设置字体是否为斜体。
- underline：设置字体是否有下划线。
- struckout：设置字体是否加删除线。
- size：字体大小。
- rotation：字体旋转角度。

下面的程序建立了多个逻辑字体：

```
static LOGFONT   *logfont, *logfontgb12, *logfontbig24;
logfont = CreateLogFont (NULL, "SansSerif", "ISO8859-1",
                         FONT_WEIGHT_REGULAR,
                         FONT_SLANT_ITALIC,
```




```
                        FONT_SETWIDTH_NORMAL,
                    FONT_SPACING_CHARCELL,
                        FONT_UNDERLINE_NONE,
                        FONT_STRUCKOUT_LINE,
                    16, 0);
logfontgb12 = CreateLogFont (NULL, "song", "GB2312",
                        FONT_WEIGHT_REGULAR,
                        FONT_SLANT_ROMAN,
                        FONT_SETWIDTH_NORMAL,
                    FONT_SPACING_CHARCELL,
                        FONT_UNDERLINE_LINE,
                        FONT_STRUCKOUT_LINE,
                    12, 0);
logfontbig24 = CreateLogFont (NULL, "ming", "BIG5",
                    FONT_WEIGHT_REGULAR,
                        FONT_SLANT_ROMAN,
                        FONT_SETWIDTH_NORMAL,
                    FONT_SPACING_CHARCELL,
                        FONT_UNDERLINE_LINE,
                        FONT_STRUCKOUT_NONE,
                    24, 0);
```

上述程序建立的字体中，logfont 是属于字符集 ISO8859-1 的字体，并且选用 SansSerif 体，字符高度为 16 像素；logfontgb12 是属于字符集 GB2312 的字体，并选用 song 体（宋体），字符高度为 12 像素；logfontbig24 是属于字符集 BIG5 的字体，并选用 ming 体（明体）。

2. 文字输出

以下函数用来输出文本（resource/minigui1.3.0/src/include/gdi.h）：

```
int GUIAPI TextOutLen (HDC hdc, int x, int y, const*  char* spText, int len);
int GUIAPI TabbedTextOutLen (HDC hdc, int x, int y, const char* spText, int len);
int GUIAPI DrawTextEx (HDC hdc, const char* pText, int nCount,
                    RECT* pRect, int nIndent, UINT nFormat);
#define TextOut(hdc, x, y, text)       TextOutLen (hdc, x, y, text, -1)
#define TabbedTextOut(hdc, x, y, text)    TabbedTextOutLen (hdc, x, y, text, -1)
#define DrawText(hdc, text, n, rc, format)    DrawTextEx (hdc, text, n, rc, 0, format)
```

TextOutLen()函数用来在给定位置输出指定长度的字符串，若长度为-1，则字符串必须是以'\0' 结尾的。TabbedTextOutLen()函数用来输出格式化字符串。DrawTextEx 是功能最复杂的输出函数，可以以不同的对齐方式在指定矩形内部输出文本。为了使用方便，MiniGUI 又定义了简化参数的对应函数，即 TextOut、TabbedTextOut 和 DrawTex。

9.7 MiniGUI 中的常用控件

同 Windows 一样，在 MiniGUI 中，程序所建立的每个窗口，都对应着某种窗口类。这一概念和面向对象编程中的类、对象的关系类似。借用面向对象的术语，MiniGUI 中的每个窗口实际都是某个窗口类的一个实例。在 X Window 编程中，也有类似的概念，比如建立的每一个 Widget，实际都是某个 Widget 类的实例。

这样，如果程序需要建立一个窗口，就首先要确保选择正确的窗口类，因为每个窗口类决定了对应窗口实例的表象和行为。这里的表象指窗口的外观，比如窗口边框宽度、是否有标题栏等；行为是指窗口对用户输入的响应。每一个 GUI 系统都会预定义一些窗口类，常见的有按钮、列表框、滚动条、编辑框等。如果程序要建立的窗口很特殊，就需要首先注册一个窗口类，然后建立这个窗口类的一个实例，这样就大大提高了代码的可重用性。

在 MiniGUI 中，主窗口被认为是一种比较特殊的窗口。因为主窗口代码的可重用性一般很低，如果按照通常的方式为每个主窗口注册一个窗口类的话，则会导致额外不必要的存储空间，所以 MiniGUI 并没有在主窗口提供窗口类支持。但主窗口中的所有子窗口，即控件，均支持窗口类（控件类）的概念。MiniGUI 提供了常用的预定义控件类，包括按钮（单选按钮、复选框）、静态框、列表框、进度条、滑块、编辑框等。程序也可以定制自己的控件类，注册后再创建对应的实例。

MiniGUI 预定义的控件类及对应类名称如表 9-3 所示。

表 9-3 MiniGUI 预定义控件类

控 件 类	类 名 称	宏 定 义
静态框	static	CTRL_STATIC
按钮	button	CTRL_BUTTON
列表框	listbox	CTRL_LISTBOX
进度条	progressbar	CTRL_PRORESSBAR
滑块	trackbar	CTRL_TRACKBAR
单行编辑框	edit、sledit	CTRL_EDIT、CTRL_SLEDIT
多行编辑框	medit、mledit	CTRL_MEDIT、CTRL_MLEDIT
工具条	toolbar	CTRL_TOOLBAR
菜单按钮	menubutton	CTRL_MENUBUTTON
树型控件	treeview	CTRL_TREEVIEW
日历控件	monthcalendar	CTRL_MONTHCALENDAR
旋钮控件	spinbox	CTRL_SPINBOX

控件的创建有两种方式：一种是在对话框模板中指定控件，这样，当应用程序启动该对话框时，系统自动创建指定控件。例如文件保存对话框、消息框等系统对话框中的按钮



和静态控件等；另一种方式是调用 CreateWindow()函数来创建控件，通过指定要生成的控件的窗口类型来生成所需控件。MiniGUI 为程序员预定义了丰富的窗口类以生成常用的窗口控件，例如静态控件、按钮、编辑框、组合框等。当然，用户也可以自己定义控件，但在使用前需向系统注册。例如，一个按钮控件的创建过程源代码如下：

```
#define IDC_CTRL1        100
...
static HWND hChildWnd1;
hChildWnd1 = CreateWindow ("BUTTON",
                           "OK",
                           WS_CHILD
                           | WS_VISIBLE,
                           IDC_CTRL1,
                           200, 180, 40, 25, hWnd, 0);
```

CreateWindow()函数在 window.h 中定义如下：

```
HWND GUIAPI CreateWindowEx (
const char * spClassName,     /*控件类型名*/
const char * spCaption,       /*控件标题*/
DWORD dwStyle,                /*控件风格*/
DWORD dwExStyle,              /*控件扩展风格*/
int id,                       /*控件标识符*/
int x,                        /*控件在窗口中的初始位置（x 坐标、y 坐标）*/
int y,
int w,                        /*控件初始大小（宽度、高度）*/
int h,
HWND hParentWnd,              /*父窗口句柄*/
DWORD dwAddData )             /*控件附加属性*/
```

应用程序创建控件后，可以通过调用 SendMessage()或 SendDlgItemMessage()函数向控件发送特定的消息来设置其属性。例如，向一进度条控件发送 PBM_SETRANGE 消息来设置其范围属性：

```
SendMessage (hChildWnd4, PBM_SETRANGE, 0, 1000);
```

各种控件都有自己特定的消息。另外，当控件的状态发生改变时（例如用户单击了按钮），控件也会向所属窗口发送相应的通知消息，把用户当前的动作情况反映给应用程序进行处理。通知消息通常是一条 MSG_COMMAND 消息，该消息的 wParam 参数的低字节为

控件标识符，高字节为控件的通知代码。

各种控件都有自己特定的风格。例如，静态控件中的文字靠左、靠右或居中分别用 SS_LEFT、SS_RIGHT 和 SS_CENTER 来标识。后面介绍各控件时将分别说明它们的专用风格。

9.7.1 静态控件与按钮控件

1. 静态控件

静态控件分为文字型和图形型两种，其中图形静态控件可以接收消息，可接收的消息为 STM_SETIMAGE 和 STM_GETIMAGE，分别用来设置和获取与图形静态控件相关联的图标或图像的句柄。

静态控件可向父窗口发送的通知代码为 STN_DBLCLK 和 STN_CLICKED，分别表示静态控件被双击或单击。

除了文字型和图片型，静态控件还有几种专用风格可供选择，如表 9-4 所示。

表 9-4 静态控件风格

风　格	描　述
SS_LEFT	控件中文字靠左对齐
SS_CENTER	控件中文字居中对齐
SS_RIGHT	控件中文字靠右对齐
SS_ICON	控件为静态图标
SS_GRAYRECT	控件为静态矩形，颜色为屏幕背景色
SS_GRAYFRAME	控件为静态边框，颜色与系统边框相同
SS_GROUPBOX	控件为分组框
SS_SIMPLE	控件为一简单矩形，其中的文字不分行且靠左对齐
SS_LEFTNOWORDWRAP	控件中的文字不分行且靠左对齐，超出部分被省略
SS_BITMAP	控件为静态图形
SS_NOPREFIX	避免将“&”符转义为快捷键标记
SS_NOTIFY	控件被单击或双击时向父窗口发送通知代码

静态控件的创建方法，其代码如下：

```
hPrompt = CreateWindow ("static",
                        "This is a static control.",
                        SS_SIMPLE | WS_VISIBLE,
                        IDC_STATIC,
                        10, 10, 185, 24, hWnd, 0);
```



2. 按钮控件

按钮控件的风格多种多样，可分为 3 大类：按键按钮、单选按钮和复选框。

- 按键按钮有两种风格：BS_PUSHBUTTON 表示标准按键按钮；BS_DEFPUSHBUTTON 表示默认按键按钮。默认按钮的特点是它在创建时就具有键盘焦点。按键按钮中还可以显示图标或图片，这时需设置其 BS_ICON 或 BS_BITMAP 属性。
- 单选按钮的风格为：BS_RADIOBUTTON 和 BS_AUTORADIOBUTTON。具有 BS_AUTORADIOBUTTON 风格的单选按钮的特点是：当几个单选按钮形成一组时，选中某个按钮的同时将使其他按钮变为未被选中状态。
- 复选框有 4 种风格：BS_CHECKBOX、BS_AUTOCHECKBOX、BS_3STATE 以及 BS_AUTO3STATE，分别表示一般复选框、自动复选框、三态复选框和自动三态复选框。

按钮控件可接收的消息如表 9-5 所示。

当用户对按钮进行操作，使按钮状态发生改变时，它就会向所属窗口发送 MSG_COMMAND 消息进行通知。表 9-6 列出了按钮通知消息中的通知代码。

表 9-5　按钮控件消息类型

消　息	描　述
BM_GETCHECK	获取单选按钮或复选框的选中状态
BM_SETCHECK	设置单选按钮或复选框的选中状态
BM_GETSTATE	获取按钮的状态
BM_SETSTATE	设置按钮的状态
BM_SETSTYLE	改变按钮风格
BM_CLICK	模拟用户单击按钮
BM_GETIMAGE	获取按钮图标或图片句柄
BM_SETIMAGE	设置新的按钮图标或图片

表 9-6　按钮的通知代码

通 知 代 码	描　述
BN_CLICKED	用户单击了按钮
BN_PUSHED	用户选中了按钮
BN_UNPUSHED	用户释放了按钮
BN_DBLCLK	用户双击了按钮
BN_SETFOCUS	按钮获得键盘焦点
BN_KILLFOCUS	按钮失去键盘焦点

下面的实例创建了 6 个不同风格的按钮控件。

```
hChildWnd1 = CreateWindow ("button",
                          "确定",
                          WS_CHILD
                          | BS_DEFPUSHBUTTON
                          | WS_VISIBLE,
                          IDC_CTRL1,
                          10, 20, 80, 30, hWnd, 0);
hChildWnd4 = CreateWindow ("button",
                          "取消",
                          WS_CHILD
                          | BS_PUSHBUTTON
                          | WS_VISIBLE,
                          IDC_CTRL4,
                          100, 20, 80, 30, hWnd, 0);
hChildWnd2 = CreateWindow ("button",
                          "复选框",
                          WS_CHILD | BS_AUTOCHECKBOX
                          | BS_LEFTTEXT | BS_RIGHT
                          | WS_VISIBLE,
                          IDC_CTRL2,
                          100, 60, 100, 30, hWnd, 0);
hChildWnd3 = CreateWindow ("button",
                          "单选钮",
                          WS_CHILD | BS_AUTORADIOBUTTON
                          | WS_VISIBLE | WS_GROUP,
                          IDC_CTRL3,
                          10, 60, 100, 30, hWnd, 0);
hChildWnd5 = CreateWindow ("button",
                          "单选钮",
                          WS_CHILD | BS_AUTORADIOBUTTON
                          | WS_VISIBLE,
                          IDC_CTRL3 + 1,
                          10, 100, 100, 30, hWnd, 0);
hChildWnd6 = CreateWindow ("button",
                          "单选钮",
                          WS_CHILD | BS_AUTORADIOBUTTON
                          | WS_VISIBLE,
                          IDC_CTRL3 + 2,
                          10, 140, 100, 30, hWnd, 0);
EnableWindow (hChildWnd6, FALSE);
```

创建的按钮控件显示效果如图 9-1 所示。





图 9-1 按钮控件创建示例

9.7.2 列表框

列表框常用风格如表 9-7 所示。

表 9-7 列表框常用风格

风 格	说 明
LBS_MULTIPLESEL	允许用户一次选择多个选项
LBS_CHECKBOX	在列表框的每一项中显示一个复选框
LBS_USEICON	在列表框的每一项中显示一个图标
LBS_SORT	列表框中的项按字母顺序排序
LBS_NOTIFY	用户操作列表框时向父窗口发送通知消息

列表框创建后，往往要添加、删除或改变列表框中的项，这些操作是通过向列表框发送相应的消息来完成的。列表框常用的消息有 LB_ADDSTRING、LB_INSERTSTRING、LB_SETCURSEL 等，详细信息请参阅 MiniGUI 函数参考手册，这里不再详细列出。

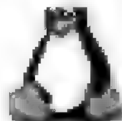
列表框通知消息中的通知代码如表 9-8 所示。

表 9-8 列表框通知代码

通 知 代 码	说 明
LBN_ERRSPACE	内存不足
LBN_SELCHANGE	用户改变列表框选择项
LBN_DBLCLK	用户双击某列表框项
LBN_SELCANCEL	用户取消对列表框项的选择
LBN_SETFOCUS	列表框项获得输入焦点
LBN_KILLFOCUS	列表框项失去输入焦点
LBN_CLICKCHECKMARK	用户单击了复选框标志
LBN_CLICKED	用户单击某列表框项

下面举例说明列表框的创建及列表框项的添加方法，示例代码如下：

```
static int ControlTestWinProc(HWND hWnd, int message, WPARAM wParam, LPARAM lParam)
{
    static HWND hChildWnd1, hChildWnd2, hChildWnd3, hChildWnd4;
    static HICON hIcon1, hIcon2;
    LISTBOXITEMINFO lbii;
    switch (message) {
        case MSG_CREATE:
            /*创建编辑对话框*/
            hChildWnd1 = CreateWindow ("Edit",
                                    "Add ListBox Item",
                                    WS_CHILD | WS_VISIBLE | WS_BORDER,
                                    IDC_CTRL1,
                                    10, 5, 190, 25, hWnd, 0);
            /*创建"添加"按钮*/
            hChildWnd2 = CreateWindow ("button",
                                    "添加",
                                    WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
                                    IDC_CTRL2,
                                    210, 5, 60, 25, hWnd, 0);
            /*创建列表框*/
            hChildWnd3 = CreateWindow ("listbox",
                                    "Listbox",
                                    WS_CHILD | WS_VISIBLE | WS_BORDER |
                                    LBS_SORT |
                                    LBS_AUTOCHECKBOX | LBS_USEICON |
                                    WS_VSCROLL,
                                    IDC_CTRL3,
                                    10, 35, 190, 100, hWnd, 0);
            /*创建"删除"按钮*/
            hChildWnd4 = CreateWindow ("button",
                                    "删除",
                                    WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON,
                                    IDC_CTRL4,
                                    210, 40, 60, 25, hWnd, 0);
            hIcon1 = LoadIconFromFile (HDC_SCREEN, "res/audio.ico", 1);
            hIcon2 = LoadIconFromFile (HDC_SCREEN, "res/cddrive.ico", 1);
            lbii.insPos = 0;
            lbii.hIcon = hIcon1;
            /*设置各列表框项的属性*/
            lbii.cmFlag = CMFLAG_CHECKED;
```




```
        lbii.string = "listitem1";
        SendMessage (hChildWnd3, LB_ADDSTRING, 0, (LPARAM)&lbii);
        lbii.cmFlag = CMFLAG_BLANK;
        lbii.string = "listitem2";
        SendMessage (hChildWnd3, LB_ADDSTRING, 0, (LPARAM)&lbii);
        lbii.cmFlag = CMFLAG_PARTCHECKED;
        lbii.string = "listitem3";
        SendMessage (hChildWnd3, LB_ADDSTRING, 0, (LPARAM)&lbii);
        lbii.cmFlag = CMFLAG_CHECKED;
        lbii.string = "listitem4";
        SendMessage (hChildWnd3, LB_ADDSTRING, 0, (LPARAM)&lbii);
    break;
    ...
```

其中的一个重要的数据结构是 LISTBOXITEMINFO，程序填充该结构以定义要操作的列表框项的属性。该结构在 MiniGUI 中定义如下(resource/minigui1.3.0/src/include/control.h)：

```
typedef struct _LISTBOXITEMINFO
{
    int      insPos;          /*列表框项插入位置*/
    char*    string;          /*显示字符串*/
    /*复选框标识，可以是如下值之一：
        CMFLAG_BLANK
        CMFLAG_CHECKED
        CMFLAG_PARTCHECKED
     */
    int      cmFlag;
    HICON    hIcon;           /*列表框项的图标句柄*/
} LISTBOXITEMINFO;
```

该代码段所创建的列表框显示效果如图 9-2 所示。



图 9-2　列表框创建示例

9.7.3 编辑控件

MiniGUI 中的编辑控件主要用于接收用户文字输入及编辑，用户可以在其中输入文字、数字以及口令等，也可以用来编辑和修改简单的文本文件。编辑控件具有的风格如表 9-9 所示。

表 9-9 编辑控件风格

风 格	说 明
ES_LEFT	控件中文字靠左对齐
ES_UPPERCASE	控件中字符自动转换为大写
ES_LOWERCASE	控件中字符自动转换为小写
ES_PASSWORD	控件中字符显示为"*"号或其他符号，用于密码输入
ES_READONLY	控件中字符为只读
ES_BASELINE	控件显示为一下划线，而不是边框
ES_AUTOWRAP	控件中字符自动换行

编辑控件能响应的消息主要有4个，如表 9-10 所示。

表 9-10 编辑控件的消息

消 息	说 明
EM_LIMITTEXT	限制控件中的字符个数
EM_SETPASSWORDCHAR	定义用于密码输入的显示字符
EM_SETREADONLY	设置字符的只读属性
EM_GETPASSWORDCHAR	获得用于密码输入的显示字符

编辑控件中，通知消息的通知代码如表 9-11 所示。

表 9-11 编辑控件通知代码

通 知 代 码	说 明
EN_CLICKED	用户单击控件
EN_DBLCLK	用户双击控件
EN_SETFOCUS	控件获得输入焦点
EN_KILLFOCUS	控件失去输入焦点
EN_CHANGE	提示控件内容变化
EN_MAXTEXT	提示控件中字符个数已达最大值
EN_ENTER	提示用户在单行编辑控件中输入了回车符



下面的代码创建了 3 个不同风格的编辑控件，第 1 个用于密码输入，第 2 个将输入字符转换为大写，第 3 个为只读编辑框。

```
HWND hWnd;
        CreateWindow (CTRL_SLEDIT,
                    "Password",
                    WS_CHILD | WS_BORDER | ES_PASSWORD | WS_VISIBLE,
                    IDC_CTRL4,
                    10, 250, 100, 24, hWnd, 0);
        CreateWindow (CTRL_EDIT,
                    "",
                    WS_CHILD | WS_BORDER | ES_UPPERCASE | WS_VISIBLE,
                    IDC_CTRL4 + 1,
                    120, 250, 100, 24, hWnd, 0);
        CreateWindow (CTRL_SLEDIT,
                    "read only box!.",
                    WS_CHILD | WS_BORDER | ES_READONLY | WS_VISIBLE,
                    IDC_CTRL5,
                    10, 310, 300, 24, hWnd, 0);
```

所创建的编辑控件显示效果如图 9-3 所示。

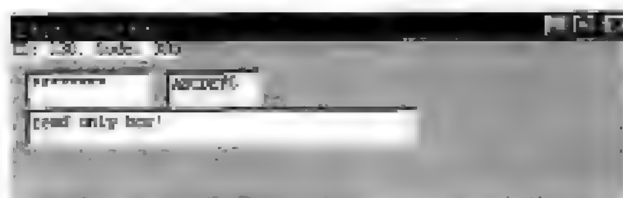


图 9-3　编辑控件创建示例

9.7.4　工具栏控件

在图形界面应用程序中，工具栏通常作为菜单命令的快捷方式，以图标方式排列于菜单栏下方，相当于一组独立命令按钮的组合。

MiniGUI 工具栏的编程方法比较简单，它只定义了一个消息 TBM_ADDITEM，用于向工具栏添加一个按钮。当然，MiniGUI 在新版本中添加了新的功能，读者可参考其源代码自行分析。下面这个例子说明工具栏的创建及按钮的添加。

```
switch (message) {
    case MSG_CREATE:
        {
```

```
            HWND    hTBSelf;
            DWORD WdHt = MAKELONG (TB_HEIGHT, TB_WIDTH);
            hTBSelf = CreateWindow ("toolbar", " ",
                            WS_CHILD | WS_VISIBLE , IDC_TB_SELF,
                            TB_BEGIN_X, TB_BEGIN_Y, HIWORD(WdHt)*6,
                            LOWORD(WdHt), hWnd, WdHt);
            InitToolBar (hTBSelf);
        }
    break;
...
static void InitToolBar (HWND toolbar)
{
    int I;
      char chtmp[MAX_PATH+1];
    TOOLBARITEMINFO pData;
    strcpy (chtmp, "res/");
    for (I = 0; I < TABLESIZE (toolbar_items); I++) {
        pData.id = toolbar_items [I].id;
        pData.insPos = I + 1;
        sprintf (pData.NbmpPath, "%s%s%s", chtmp, toolbar_items [I].name, "1.bmp");
        sprintf (pData.HbmpPath, "%s%s%s", chtmp, toolbar_items [I].name, "2.bmp");
        sprintf (pData.DbmpPath, "%s%s%s", chtmp, toolbar_items [I].name, "3.bmp");
        SendMessage (toolbar, TBM_ADDITEM, 0, (LPARAM)&pData);
    }
}
```

其中数据结构 TOOLBARITEMINFO 保存工具栏按钮项的属性，程序填充该结构以定义要操作的工具栏的各个属性。该结构在 MiniGUI 中定义如下（resource/minigui1.3.0/src/include/control.h）：

```
typedef struct _TOOLBARITEMINFO
{
    int     insPos;  /*插入项位置*/
    int     id;             /*插入项 ID*/
    /*插入项正常状态显示图标的文件路径*/
    char    NbmpPath[MAX_PATH+10];
    /*插入项高亮状态显示图标的文件路径*/
    char    HbmpPath[MAX_PATH+10];
    /*插入项按下状态显示图标的文件路径*/
    char    DbmpPath[MAX_PATH+10];
    /*附加字段*/
```




```
    DWORD   dwAddData;
} TOOLBARITEMINFO;
```

9.7.5 控件子类化

采用控件类和控件实例的结构，不仅可以提高代码的可重用性，而且还可以方便地对已有控件类进行扩展。比如，在需要建立一个只允许输入数字的编辑框时，就可以通过重载已有编辑框控件类而实现，而不需要重新编写一个新的控件类。在 MiniGUI 中，这种技术称为子类化或者窗口派生。子类化的方法有以下 3 种：

- 对已经建立的控件实例进行子类化，子类化的结果只影响这一个控件实例。
- 对某个控件类进行子类化，子类化的结果将影响其后创建的所有该控件类的控件实例。
- 在某个控件类的基础上新注册一个子类化的控件类，子类化的结果不会影响原有控件类。在 Windows 中，这种技术又称为超类化。

在 MiniGUI 中，控件的子类化实际是通过替换已有的窗口过程实现的。下面的示例代码就是通过控件类创建了两个子类化的编辑框，一个只能输入数字，而另一个只能输入字母。

```
#define IDC_CTRL1       100
#define IDC_CTRL2       110
#define IDC_CTRL3       120
#define IDC_CTRL4       130

#define MY_ES_DIGIT_ONLY     0x0001
#define MY_ES_ALPHA_ONLY     0x0002
static WNDPROC old_edit_proc;
static int RestrictedEditBox (HWND hwnd, int message, WPARAM wParam, LPARAM lParam)
{
    if (message == MSG_CHAR) {
        DWORD my_style = GetWindowAdditionalData (hwnd);
        /*确定被屏蔽的按键类型*/
        if ((my_style & MY_ES_DIGIT_ONLY) && (wParam < '0' || wParam > '9'))
            return 0;
        else if (my_style & MY_ES_ALPHA_ONLY)
            if (!((wParam >= 'A' && wParam <= 'Z') || (wParam >= 'a' && wParam <= 'z')))
                /*收到被屏蔽的按键消息，直接返回*/
                return 0;
    }
    /*由老的窗口过程处理其余消息*/
    return (*old_edit_proc) (hwnd, message, wParam, lParam);
```

```
}

static int ControlTestWinProc (HWND hWnd, int message, WPARAM wParam, LPARAM
lParam)
{
  switch (message) {
  case MSG_CREATE:
  {
    HWND hWnd1, hWnd2, hWnd3;
    CreateWindow (CTRL_STATIC, "Digit-only box:",
                            WS_CHILD | WS_VISIBLE | SS_RIGHT, 0,
                        10, 10, 180, 24, hWnd, 0);
    hWnd1 = CreateWindow (CTRL_EDIT, "",
                            WS_CHILD | WS_VISIBLE | WS_BORDER, IDC_CTRL1,
                        200, 10, 180, 24, hWnd, MY_ES_DIGIT_ONLY);
    CreateWindow (CTRL_STATIC, "Alpha-only box:",
                            WS_CHILD | WS_VISIBLE | SS_RIGHT, 0,
                        10, 40, 180, 24, hWnd, 0);
hWnd2 = CreateWindow (CTRL_EDIT, "",
                            WS_CHILD | WS_BORDER | WS_VISIBLE, IDC_CTRL2,
                        200, 40, 180, 24, hWnd, MY_ES_ALPHA_ONLY);
    CreateWindow (CTRL_STATIC, "Normal edit box:",
                            WS_CHILD | WS_VISIBLE | SS_RIGHT, 0,
                        10, 70, 180, 24, hWnd, 0);
    hWnd3 = CreateWindow (CTRL_EDIT, "",
                            WS_CHILD | WS_BORDER | WS_VISIBLE, IDC_CTRL2,
                        200, 70, 180, 24, hWnd, MY_ES_ALPHA_ONLY);
    CreateWindow ("button", "Close",
                            WS_CHILD | BS_PUSHBUTTON
                            | WS_VISIBLE, IDC_CTRL4,
                        100, 100, 60, 24, hWnd, 0);
    /*用自定义的窗口过程替换编辑框的窗口过程，并保存老的窗口过程*/
    old_edit_proc = SetWindowCallbackProc (hWnd1, RestrictedEditBox);
    SetWindowCallbackProc (hWnd2, RestrictedEditBox);
    break;
  }
...
}
return DefaultMainWinProc (hWnd, message, wParam, lParam);
}
```

程序首先定义了一个窗口处理过程，即 RestrictedEditBox()函数。然后，在利用



CreateWindow()函数建立控件时，将其中两个编辑框的窗口处理过程通过SetWindowCallbackProc替换成了自己定义的RestrictedEditBox()函数，并且将该函数返回的值（即老的控件窗口处理过程地址）保存在old_edit_box变量中。在建立这些编辑框之后，它们的消息将首先由RestrictedEditBox()函数处理，然后在某些情况下才由老的窗口处理过程处理。

限于篇幅，另外两种控件子类化的方法就不在这里讲述了。

9.7.6 自定义控件

用户也可以通过RegisterWindowClass()函数注册自己的控件类，并建立该控件类的控件实例。如果程序不再使用某个自定义的控件类，则应该使用UnregisterWindowClass()函数注销自定义的控件类。

RegisterWindowClass()函数通过pWndClass结构注册一个控件类；UnregisterWindowClass()函数则注销指定的控件类；GetClassName()函数获得窗口的对应窗口类名称，对主窗口而言，窗口类名称为"MAINWINDOW"；GetWindowClassInfo()函数用来获取和指定特定窗口类的属性。

下面的示范程序定义并注册了一个自己的控件类。该控件用来显示安装程序的步骤信息：MSG_SET_STEP_INFO消息用来定义该控件中显示的所有步骤信息，包括所有步骤名称及其简单描述；MSG_SET_CURR_STEP消息用来指定当前步骤，控件将高亮显示当前步骤。

```
#define STEP_CTRL_NAME "mystep"
#define MSG_SET_STEP_INFO      (MSG_USER + 1)
#define MSG_SET_CURR_STEP      (MSG_USER + 2)
static int StepControlProc (HWND hwnd,
                    int message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    HELPWININFO* info;

    switch (message) {
    case MSG_PAINT:
        hdc = BeginPaint (hwnd);
        /*获取步骤控件信息*/
        info = (HELPWININFO*)GetWindowAdditionalData (hwnd);
        /*绘制步骤内容*/
        ......
        EndPaint (hwnd, hdc);
        break;
```

```
    /*控件自定义的消息：用来设置步骤信息*/
    case MSG_SET_STEP_INFO:
        SetWindowAdditionalData (hwnd, (DWORD)lParam);
        InvalidateRect (hwnd, NULL, TRUE);
        break;
    /*控件自定义的消息：用来设置当前步骤信息*/
    case MSG_SET_CURR_STEP:
        InvalidateRect (hwnd, NULL, FALSE);
        break;
    case MSG_DESTROY:
        break;
    }
    return DefaultControlProc (hwnd, message, wParam, lParam);
}
static BOOL RegisterStepControl ()
{
    int result;
    WNDCLASS StepClass;
    StepClass.spClassName = STEP_CTRL_NAME;
    StepClass.dwStyle       = 0;
    StepClass.hCursor       = GetSystemCursor (IDC_ARROW);
    StepClass.iBkColor      = COLOR_lightwhite;
    StepClass.WinProc       = StepControlProc;
    return RegisterWindowClass (&StepClass);
}
static void UnregisterStepControl ()
{
    UnregisterWindowClass (STEP_CTRL_NAME);
}
```

9.8 对话框

对话框编程是一个快速构建用户界面的技术。通常，编写简单的图形用户界面时，可以通过调用 CreateWindow()函数直接创建所有需要的子窗口，即控件。但在图形用户界面比较复杂的情况下，每建立一个控件就调用一次 CreateWindow()函数，并传递许多复杂参数的方法很不可取。主要原因之一就是程序代码和用来建立控件的数据混在一起，不利于维护。为此，一般的 GUI 系统都会提供一种机制，利用这种机制，通过指定一个模板 GUI 系统就可以根据此模板建立相应的主窗口和控件。MiniGUI 也提供这种方法，通过建立对话框模板，就可以建立模式或者非模式的对话框。



9.8.1 创建模式对话框

模式对话框指的是显示之后用户不能再切换到其他主窗口进行工作的对话框，而只能在关闭之后，才能使用其他的主窗口。在 MiniGUI 中，使用 DialogBoxIndirectParam()函数建立的对话框就是模式对话框。实际上，该对话框首先根据模板建立对话框，然后禁止其托管主窗口，并在主窗口的 MSG_CREATE 消息中创建控件，并发送 MSG_INITDIALOG 消息给回调函数，最终建立一个新的消息循环，并进入该消息循环，直到程序调用 EndDialog()函数为止。

下面的示例程序示范如何建立一个输入新密码的对话框：

```
/*模式对话框创建函数*/
static void testDialogBox (HWND hWnd)
{
    DlgPassword.controls = CtrlPassword;
    DialogBoxIndirectParam (&DlgPassword, hWnd, DialogBoxProc3, 0L);
}
/*对话框控件定义*/
CTRLDATA CtrlPassword [] =
{
    {
        "static",
        WS_VISIBLE | SS_RIGHT,
        14, 30, 150, 18,
        IDC_STATIC,
        "请输入新的口令",
        0
    },
    {
        "edit",
        WS_CHILD | WS_VISIBLE | WS_BORDER | ES_PASSWORD | WS_TABSTOP,
        180, 30, 200, 24,
        IDC_NEWPASSWORD,
        NULL,
        0
    },
    {
        "button",
        WS_VISIBLE | BS_AUTOCHECKBOX | WS_TABSTOP,
        180, 70, 100, 22,
```

```
        IDC_PASSWORDVALID,
        "密码有效",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
        80, 94, 100, 28,
        IDOK,
        "确定",
        0
    },
    {
        "button",
        WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
        276, 94, 100, 28,
        IDCANCEL,
        "取消",
        0
    }
};
/*对话框消息循环函数*/
DialogBoxProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_INITDIALOG:
        return 1;
    case MSG_COMMAND:
        switch (wParam) {
        case IDOK:
        case IDCANCEL:
            EndDialog (hDlg, wParam);
            break;
        }
        break;

    }
    return DefaultDialogProc (hDlg, message, wParam, lParam);
}
```

该对话框显示效果如图 9-4 所示。

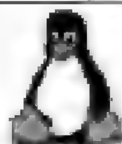




图 9-4　设置新密码对话框

9.8.2　创建非模式对话框

MiniGUI 使用 CreateMainWindowIndirect()函数来创建非模式对话框。实际上，这是利用对话框模板建立普通的主窗口。使用 CreateMainWindowIndirect()函数根据对话框模板建立的主窗口和其他类型的普通主窗口没有任何区别。非模式对话框的创建过程与模式对话框的创建过程基本相同，这里不再详述。

9.8.3　带属性页的对话框

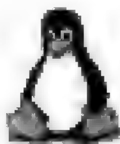
对于操作对象较多的对话框，使用属性页对操作进行分组可以使整个界面简洁有序，因此，图形界面应用程序中往往大量使用带属性页的对话框。

属性页对话框的创建与一般的对话框相似，只是属性页的创建较为复杂。创建过程分为两个步骤：首先，定义各属性页的控件资源及消息循环函数，并在对话框初始化阶段创建属性页控件；然后将该属性页控件定义为对话框的控件资源，调用 DialogBoxIndirectParam()函数创建对话框。

下面的示范程序创建了一个带有两个属性页的对话框：

```
/*属性页对话框创建函数*/
void testPropertySheet (HWND hWnd)
{
    DlgPropertySheet.controls = CtrlPropertySheet;
    DialogBoxIndirectParam (&DlgPropertySheet, hWnd, PropSheetProc, 0L);
}
/*对话框控件资源定义*/
CTRLDATA CtrlPropertySheet [] =
{
    {
        CTRL_PROPSHEET,
        WS_VISIBLE | WS_TABSTOP | PSS_COMPACTTAB,
        10, 10, 560, 360,
```

```
            IDC_PROPSHEET,
            "Property sheet control",
            0
        },
        {
            "button",
            WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
            200, 380, 100, 28,
            IDC_APPLY,
            "应用",
            0
        },
        {
            "button",
            WS_VISIBLE | BS_DEFPUSHBUTTON | WS_TABSTOP,
            340, 380, 100, 28,
            IDOK,
            "确定",
            0
        },
        {
            "button",
            WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
            460, 380, 100, 28,
            IDCANCEL,
            "取消",
            0
        }
};
/*对话框消息处理函数*/
static int PropSheetProc (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case MSG_INITDIALOG:
        {
            HWND pshwnd = GetDlgItem (hDlg, IDC_PROPSHEET);
            DlgPassword.controls = CtrlPassword;
            SendMessage ( pshwnd, PSM_ADDPAGE,
                        (WPARAM)&DlgPassword,(LPARAM) PageProc1);
            DlgInitProgress.controls = CtrlInitProgress;
            SendMessage ( pshwnd, PSM_ADDPAGE,
                        (WPARAM)&DlgInitProgress, (LPARAM) PageProc2);
```




```
                break;
            }
            case MSG_COMMAND:
            switch (wParam)
            {
                case IDC_APPLY:
                break;
                case IDOK:
                {
                    int index = SendDlgItemMessage (hDlg, IDC_PROPSHEET,
                                    PSM_SHEETCMD, IDOK, 0);
                    if (index) {
                        SendDlgItemMessage (hDlg, IDC_PROPSHEET,
                                    PSM_SETACTIVEPAGE, index - 1, 0);
                    }
                    else
                        EndDialog (hDlg, wParam);
                    break;
                }
                case IDCANCEL:
                    EndDialog (hDlg, wParam);
                break;
            }
            break;
        }
        return DefaultDialogProc (hDlg, message, wParam, lParam);
}
/*属性页控件资源*/
CTRLDATA CtrlPassword [] =
{
    {
        "static",
        WS_VISIBLE | SS_RIGHT,
        14, 30, 150, 18,
        IDC_STATIC,
        "请输入新的口令",
        0
    },
    {
        "edit",
        WS_CHILD | WS_VISIBLE | WS_BORDER | ES_PASSWORD | WS_TABSTOP,
        180, 30, 200, 24,
```

```
            IDC_NEWPASSWORD,
            NULL,
            0
    },
    {
            "button",
            WS_VISIBLE | BS_AUTOCHECKBOX | WS_TABSTOP,
            180, 70, 100, 22,
            IDC_PASSWORDVALID,
            "密码有效",
            0
    },
    {
            "button",
            WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
            80, 94, 100, 28,
            IDOK,
            "确定",
            0
    },
    {
            "button",
            WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
            276, 94, 100, 28,
            IDCANCEL,
            "取消",
            0
    }
};
CTRLDATA CtrlInitProgress [] =
{
    {
            "static",
            WS_VISIBLE | SS_SIMPLE,
            10, 10, 380, 16,
            IDC_PROMPTINFO,
            "当前屏幕分辨率为:",
            0
    },
    {
            "edit",
            WS_VISIBLE | WS_BORDER | WS_TABSTOP,
```




```
            10, 40, 200, 20,
            IDC_DIF,
            NULL,
            0
        },
        {
            "button",
            WS_TABSTOP | WS_VISIBLE | BS_DEFPUSHBUTTON,
            170, 70, 60, 25,
            IDOK,
            "确定",
            0
        }
};
/*属性页消息处理函数*/
static int
PageProc1 (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_INITPAGE:
        break;
    case MSG_SHOWPAGE:
        return 1;
    case MSG_COMMAND:
        switch (wParam) {
        case IDOK:
        case IDCANCEL:
            MessageBox (hDlg, "Button pushed", "OK",
                                MB_OK | MB_ICONINFORMATION |
                                    MB_BASEDONPARENT);
            break;
        }
        break;
    }
    return DefaultPageProc (hDlg, message, wParam, lParam);
}
static int
PageProc2 (HWND hDlg, int message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
    case MSG_INITPAGE:
        break;
```

```
    case MSG_SHOWPAGE:
        return 1;
    case MSG_COMMAND:
        switch (wParam) {
        case IDOK:
        case IDCANCEL:
            MessageBox (hDlg, "Button pushed", "OK",
                        MB_OK | MB_ICONINFORMATION |
                        MB_BASEDONPARENT);
            break;
        }
    }
    return DefaultPageProc (hDlg, message, wParam, lParam);
}
```

显示效果如图 9-5 所示。

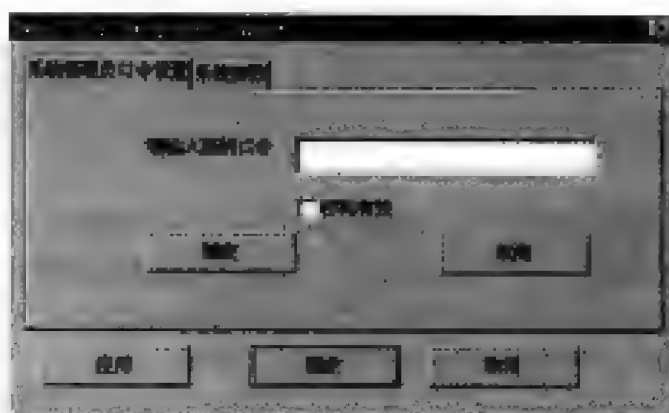


图 9-5 带属性页的对话框

9.9 菜单的使用

菜单是图形界面应用程序的重要组件，用户通过菜单可以方便地选择程序的命令及选项。菜单有两种类型：一种是主菜单，位于标题栏的下方；另一种是弹出式菜单，由用户右击激活，一般在鼠标当前位置弹出显示。

9.9.1 创建菜单

1. 主菜单的创建

MiniGUI 提供了一个菜单结构 MENUITEMINFO 及 3 个 API 函数，用于菜单创建。应



用程序在 MENUITEMINFO 结构中保存各个菜单项的属性，结构定义如下：

```
typedef struct _MENUITEMINFO {
    UINT    mask;                          /*菜单项信息掩码*/
                                           /*菜单项类型，可以是如下值*/
     MFT_STRING
     MFT_BITMAP
     MFT_BMPSTRING
     MFT_SEPARATOR
     MFT_RADIOCHECK*/
    UINT    type;
    UINT    state;                         /*菜单项状态*/
    int     id;                            /*菜单项 ID 号*/
    HMENU       hsubmenu;                  /*子菜单句柄*/
    PBITMAP     hbmpChecked;               /*选中菜单项的位图对象指针*/
    PBITMAP     hbmpUnchecked;             /*未选中菜单项的位图对象指针*/
    DWORD           itemdata;              /*私有数据*/
    DWORD           typedata;              /*显示字串*/
    UINT            cch;                   /*字串最大长度*/
} MENUITEMINFO;
```

3 个相关的 API 函数分别为 CreateMenu()、CreatePopupMenu()和 InsertMenuItem()。CreateMenu()函数用于创建一个空白的主菜单，CreatePopupMenu()函数用于创建空白弹出式菜单或子菜单，而 InsertMenuItem()函数用于向菜单中添加菜单项或是向主菜单中添加子菜单。利用这 3 个函数，应用程序依以下步骤创建主菜单：

（1）构建子菜单创建函数，函数中调用 CreatePopupMenu()函数创建空白子菜单，调用 InsertMenuItem()函数向子菜单中插入菜单项。

（2）构建主菜单创建函数，函数中调用 CreateMenu()函数创建主菜单，将其 MENUITEMINFO 结构的子菜单句柄指向前面的子菜单创建函数，而后调用 InsertMenuItem()函数将子菜单添加到主菜单中。

（3）在程序入口函数中将 MAINWINCREATE 结构的菜单句柄指向前面创建的主菜单对象。

下面的代码示范了主菜单的创建过程：

```
/*子菜单创建函数*/
static HMENU createpmenuedit (void)
{
    HMENU hmnu;
    MENUITEMINFO mii;
    memset (&mii, 0, sizeof(MENUITEMINFO));
```

```
    mii.type          = MFT_STRING;
    mii.id            = 0;
    mii.typedata      = (DWORD)"Edit";
    hmnu = CreatePopupMenu (&mii);

    mii.type          = MFT_STRING ;
    mii.state         = 0;
    mii.id            = IDM_COPY;
    mii.typedata      = (DWORD)"Copy Screen";
    InsertMenuItem(hmnu, 0, TRUE, &mii);

    return hmnu;
}
/*主菜单创建函数*/
static HMENU createmenu (void)
{
    HMENU hmnu;
    MENUITEMINFO mii;

    hmnu = CreateMenu();

    mii.type          = MFT_STRING;
    mii.id            = 110;
    mii.typedata      = (DWORD)"Edit";
    mii.hsubmenu      = createpmenuedit ();
    InsertMenuItem(hmnu, 1, TRUE, &mii);

    return hmnu;
}
/* MAINWINCREATE 结构初始化函数*/
static void InitCreateInfo (PMAINWINCREATE pCreateInfo)
{
    pCreateInfo->dwStyle = WS_CAPTION | WS_VISIBLE;
    pCreateInfo->dwExStyle = 0;
    pCreateInfo->spCaption = "MiniGUI Demo";
    pCreateInfo->hMenu = createmenu();
    pCreateInfo->hCursor = GetSystemCursor (IDC_PENCIL);
    pCreateInfo->hIcon = 0;
    pCreateInfo->MainWindowProc = DemoWinProc;
    pCreateInfo->lx = 0;
    pCreateInfo->ty = 0;
    pCreateInfo->rx = pCreateInfo->lx + DEFAULT_WIDTH;
```




```
    pCreateInfo->by = pCreateInfo->ty + DEFAULT_HEIGHT;
    pCreateInfo->iBkColor = PIXEL_lightwhite;
    pCreateInfo->dwAddData = 0;
    pCreateInfo->hHosting = HWND_DESKTOP;
}
```

上面的示范函数创建出来的主菜单非常简单，只有一个“Edit”项，其子菜单也只有一个“Copy Screen”菜单项。

2. 弹出式菜单的创建

弹出式菜单平时并不显示，只在用户需要时在特定位置（一般是光标当前位置）弹出。MiniGUI 用 TrackPopupMenu()函数来显示并跟踪一个弹出式菜单，该函数定义如下：

int GUIAPI TrackPopupMenu (HMENU hmnu, UINT uFlags, int x, int y, HWND hwnd);

其中参数 hmnu 为弹出菜单句柄；uFlags 为跟踪标记，定义弹出菜单的响应模式；x、y 为菜单弹出位置的屏幕坐标；hwnd 为接收菜单消息的窗口句柄。下面的程序段示范了弹出式菜单的创建方法：

```
x = LOSWORD(up->lParam);
y = HISWORD(up->lParam);                              /*鼠标位置坐标*/
ClientToScreen (hChildWnd1, &x, &y);                  /*将鼠标位置转换为屏
                                                        幕坐标*/
TrackPopupMenu (
            GetPopupSubMenu (hRightMenu),             /*弹出菜单句柄*/
            TPM_LEFTALIGN | TPM_LEFTBUTTON ,          /*跟踪标记*/
            x, y,                                     /*菜单弹出位置屏幕
                                                        坐标*/
            hMainWnd);                                /*所属窗口句柄*/
```

9.9.2 处理菜单消息

菜单创建后，要让用户在选中某个菜单项后得到正确的响应，就必须为每个菜单项定义动作。与控件相同，菜单项被单击后，也是通过发送 MSG_COMMAND 消息通知主窗口，主窗口消息处理函数依据该消息的 wParam 参数区分发出消息的菜单项，从而对菜单消息作出正确响应。

下面的代码表示程序对用户选中“Copy Screen”项作出响应，弹出一个提示框：

```
case MSG_COMMAND:
        switch (wParam)
        {
```

```
    ...
case IDM_COPY:
    MessageBox (hWnd,
        "您选择了 "Copy Screen" 项",
        " MiniGUI Demo",
        MB_OK | MB_ICONINFORMATION |
         MB_BASEDONPARENT);
break;
    ...
```

9.9.3 更改菜单项状态

菜单项可以有多种状态，例如，应用程序可根据当前环境将某些菜单项设置为有效、无效或灰色，以控制用户访问权限。MiniGUI 提供 EnableMenuItem()函数来改变菜单项的有效状态，该函数原型如下：

UINT GUIAPI EnableMenuItem (HMENU hmnu, int item, UINT flag);

以下是参数说明：

- hmnu：菜单对象句柄。
- item：菜单项位置。
- flag：指示菜单项位置是基于命令标识符（MF_BYCOMMAND），还是基于菜单项在菜单中的序号（MF_BYPOSITION）。

下面的代码使一个菜单项变为灰色：

```
EnableMenuItem (hmnu,IDM_COPY, MF_BYCOMMAND);
```

菜单项还有选中和未选中状态，选中状态下的菜单项前面会有一个圆点（•）标识，以提醒用户当前应用程序所处状态。MiniGUI 提供 CheckMenuRadioItem()函数来改变菜单项的选中和未选中状态，该函数原型如下：

```
int GUIAPI CheckMenuRadioItem (HMENU hmnu, int first, int last, int checkitem, UINT flag);
```

以下是参数说明：

- hmnu：菜单对象句柄。
- first：菜单项组的第一项所在位置。
- last：菜单项组的最后一项所在位置。
- checkitem：选中项所在位置。
- flag：指示菜单项位置是基于命令标识符（MF_BYCOMMAND），还是基于菜单项在菜单中的序号（MF_BYPOSITION）。

提示："first" 菜单项和 "last" 菜单项共同定义出一个菜单项组。CheckMenuRadioItem() 函数使其中的某项选中，同时自动使该组中的其他项变为未选中状态。

下面的代码使某一菜单项被选中：

```
CheckMenuRadioItem ((HMENU)lParam,
                    IDM_DEFAULT, IDM_BIG5,
                    pNoteInfo->editCharset, MF_BYCOMMAND);
```

9.10 小结

MiniGUI 是一个优秀的自由软件，已经被广泛应用于许多项目和产品中。本章介绍嵌入式 Linux 环境下的 MiniGUI 图形界面应用程序开发，分析了 MiniGUI 的安装配置以及各图形界面元素的具体编程方法与技巧。当然，限于本书的定位，本章没有也不可能对它进行面面俱到的分析，而只是选出其中比较重要且具有代表性的部分来介绍。通过这些介绍，读者应该可以掌握 MiniGUI 环境下的图形界面应用程序开发的基本技术。MiniGUI 的项目主持者——北京飞漫软件技术有限公司提供了完善的产品培训及技术支持，如果读者有这方面的项目需求，可以购买产品以获得全面指导。

9.11 思考题

1. 到 MiniGUI 网站 www.minigui.com 下载其最新版本，在自己的机器上安装并正确配置。
2. 分析 MiniGUI 的 Threads 版本和 Lite 版本的区别，总结它们各自的适用领域。
3. 分析 MiniGUI 程序的 C/S 运行模式，考虑如何创建自己的 mginit 服务器程序。
4. 比较 MiniGUI 与 Windows 系统对各图形界面元素的实现方法的异同点。
5. 编写一个控件使用演示程序，创建出常用的控件，并练习创建控件时的位置与大小控制，使界面尽量整齐美观。
6. 参考 Windows 系统中对自画控件的实现，创建一个图片按钮。



第 10 章　USB 设备驱动程序开发

知识点：

- USB 原理与协议
- Linux 环境下 USB 驱动程序结构
- USB 驱动程序数据结构与函数

本章导读：

本章介绍嵌入式 Linux 环境下 USB 设备驱动程序开发的基础知识。首先介绍 USB 原理及其协议，使读者了解进行 USB 驱动程序开发的基本要素。而后，详细介绍 USB 设备驱动程序的分类及各类驱动程序的框架结构、内核支持函数、数据结构及宏定义等，并具体分析 USB 各层协议的实现，对其中的重要环节给出示例代码。



10.1 USB 体系结构

USB（Universal Serial Bus）即通用串行总线，它是一种全新的、双向同步传输的、支持热插拔的数据传输总线，由 Compaq、Intel、Microsoft 以及 NEC 等公司共同开发。其目的是为了提供一种兼容低速和高速的、可扩充并且使用方便的外围设备接口，同时也为了解决计算机接口太多的弊端。现在，USB 接口已被人们广泛接受，越来越多的 USB 产品不断涌现出来。

10.1.1 USB 系统的描述

一个 USB 系统主要由 3 个部分组成：

- USB 互连。
- USB 主机。
- USB 设备。

USB 互连是指 USB 设备与主机之间进行连接和通信的操作，主要包括总线的拓扑结构、数据流模式、USB 的调度等。USB 主机及设备的相关知识将在后续章节中加以介绍。

1. 总线布局

USB 连接了 USB 设备和 USB 主机，USB 的物理连接是层叠的星型结构，如图 10-1 所示。每个星型的中心是一个集线器，星形的每个点就是一个连接到集线器的某个端口的设备，这些设备有可能是其他集线器或外设。

但是在编程时，不必关心它的物理连接。总线上的所有设备共享一条通往主机的数据通道，编程时只需认为某个外设是独占 USB 端口的。

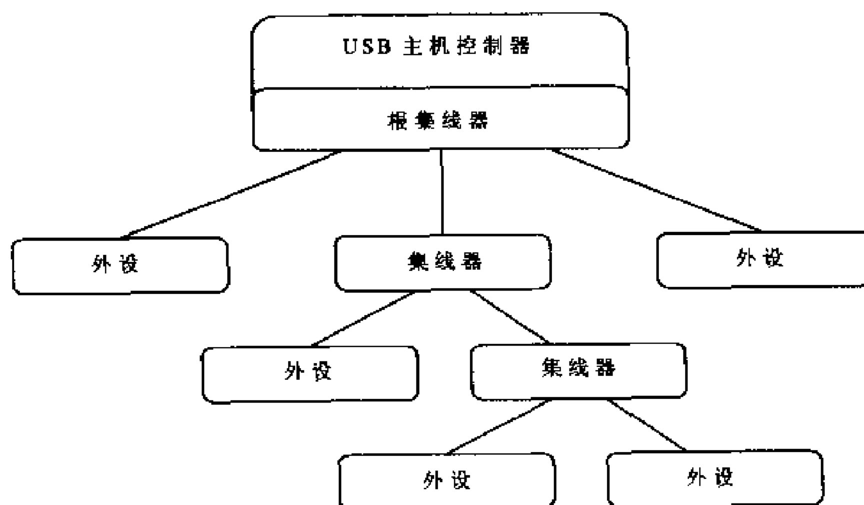
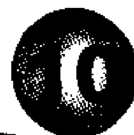


图 10-1 USB 系统拓扑结构



2. USB 主机

一个 USB 系统中只有一个主机。USB 和主机系统的接口称做主机控制器，主机控制器可由硬件、固件和软件综合实现。根集线器是由主机系统整合的，用以提供更多的连接点。

3. USB 设备

USB 的设备如下所示：

- 网络集线器：向 USB 提供了更多的连接点。
- 功能器件：为系统提供具体功能，如 ISDN 的连接、数字的游戏杆或扬声器。

USB 设备提供的 USB 标准接口的主要依据：

- 对 USB 协议的运用。
- 对标准 USB 操作的反馈，如设置和复位。
- 标准性能的描述性信息。

10.1.2 电气特性

USB 总线通过一根四线电缆传送信号和电源，图 10-2 中的 D+和 D-两根线用于发送信号。

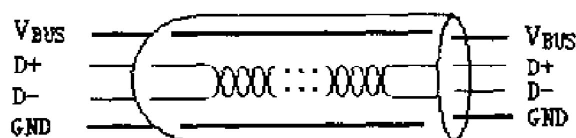


图 10-2 SB 电缆

USB 总线存在 3 种数据传输速率：

- 高速信号传送的比特率定为 480Mbit/s。
- 全速信号传送的比特率定为 12Mbit/s。
- 低速信号传送的比特率定为 1.5Mbit/s。

低速模式需要更少的 EMI 保护，3 种数据传输模式可在同一 USB 总线传输的情况下自动地动态切换。因为过多的低速模式的使用将降低总线的利用率，所以低速模式只支持有限个低带宽的设备（如鼠标）。时钟被调制后与差分数据一同被传送出去，时钟信号被转换成 NRZI 码，并填充了比特以保证转换的连续性，每一数据包中附有同步信号以使接收方还原出原时钟信号。

电缆中包括 VBUS、GND 两条线向设备提供电源。VBUS 使用+5V 电源，USB 对电缆长度的要求很宽，最长可为几米。通过选择合适的导线长度以匹配指定的 IR drop 和其他一些特性，如设备能源预算和电缆适应度。为了保证足够的输入电压和终端阻抗，重要的终端设备应位于电缆的尾部。在每个端口都可检测终端是否连接或分离，并区分出高速或低速设备。



10.1.3 电源分配与管理

1. 电源分配

每个 USB 单元通过电缆只能提供有限的能源。主机对直接相连的 USB 设备提供电源供其使用，并且每个 USB 设备都可能有自己的电源。那些完全依靠电缆提供能源的设备称为“总线供能”设备。相反，那些可选择能源来源的设备称为“自供电”设备。而且，集线器也可由与之相连的 USB 设备提供电源。

2. 电源管理

USB 主机拥有一个独立于 USB 总线的电源管理系统。USB 的系统软件可以与主机的电源管理系统结合共同处理挂起或恢复等系统电源事件。另外，USB 设备也可以由系统软件对其进行电源管理。USB 合理的电源分配和电源管理特性使之可以被应用在低功耗系统中，如采用电池的笔记本电脑或其他手持移动设备如 PDA 等。

10.2 USB 通信协议

10.2.1 USB 数据流模型

图 10-3 是 USB 的通信模型示意图，从图 10-3 中可以看出 USB 通信的数据流的结构。主机的每一个层次分别对应设备相应的层次，通过逻辑通道连接起来，客户软件通过逻辑连接可以直接控制设备的接口模块。这种连接使得软件控制与接口一一对应起来，用户使用起来可以更加简单快捷。

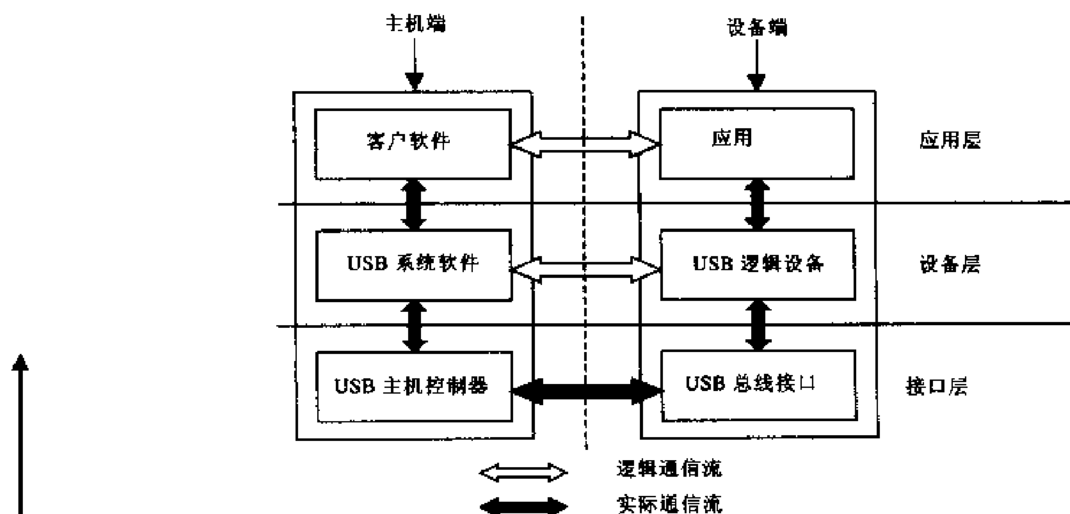
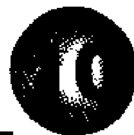


图 10-3 USB 通信模型



USB 总线有 4 种传输方式，分别是控制传输、中断传输、批量传输和同步传输。

- 控制传输

主要用于主机向 USB 设备发送命令或是 USB 设备将自身状态信息返回给主机。任何一个 USB 设备必须支持一个与控制类型相对应的端点 0，即主控端点。

控制传输在所有 USB 设备中都需要使用，主机对 USB 设备的配置命令需通过控制传输来发送，而设备的描述信息也需通过控制传输传递给主机。

- 中断传输

用来支持那些偶然需要少量数据通信，但服务时间受限制的设备。中断传输常常用在键盘、鼠标、游戏杆等外设上。

- 批量传输

主要用于需要传输大量数据，且数据传输速率不固定的设备。批量传输方式并不能保证传输的速率，但可保证传输的可靠性，当出现错误时会要求发送方重发。

- 同步传输

同步传输要求有一个恒定的速率。同步传输方式的发送方和接收方都必须保证传输速率的匹配，不然会造成数据的丢失。这种传输方式适用于实时性要求高而对准确性要求较低的场合，如音频和视频设备、语音和图像不能有明显滞后，但中间的某些数据错误则关系不大，不影响人的视觉和听觉。

10.2.2 USB 数据单元

对应主机客户软件与设备应用间的不同通信服务，USB 设备对数据流有不同的要求。为此，USB 允许各种不同的数据流相互独立地进入一个 USB 设备，不同设备的不同端点用于区分不同的数据流。

1. 端点（Endpoint）

端点是 USB 设备的逻辑设备。一个 USB 设备有一个或多个端点，对主机而言，它们对应一个或多个逻辑设备。设备连上主机时，主机分配给每个逻辑设备一个惟一地址，而设备中的每个端点在设备内部有一个惟一的端点号，由设备设计时指定。这样，主机与设备间就可以依据端点来建立惟一通道。

所有 USB 设备至少要有一个 0 号端点，USB 系统使用这个默认端点设置设备。对于低速设备，除 0 号端点外，只能有 2 个额外的可选端点。而全速设备和高速设备则可以最多有 15 个额外输入端点和 15 个输出端点。

2. 通道（Pipe）

设备上的端点和主机上的逻辑设备地址构成一个 USB 通道，主机与设备间的信息交互就是通过通道进行的。

依据通信数据格式的不同，通道分为以下两种：

- 流（Stream）：指不具有 USB 定义的格式的数据流。



● 消息（Message）：指具有某种 USB 定义的格式的数据流。

消息通道要求把数据组织成 USB 定义的格式。

3. 字段（Field）

USB 的传输以包为单位，每个包由不同的字段组成。主要有以下几种字段：

● 同步字段（SYNC）

所有的包都是以同步字段开始，输入电路根据同步字段以本地时钟对齐输入数据。同步字段的最后 2 位是字段结束标记，同时标志包标识符（PID，Packet Identifer）字段的开始。

● 包标识符字段（PID）

USB 包的同步字段后紧跟着包标识符字段。包标识符由 4 位包类型字段及其后的 4 位校验字段构成，如图 10-4 所示。包标识符指出包的类型，校验字段起一个检错作用。

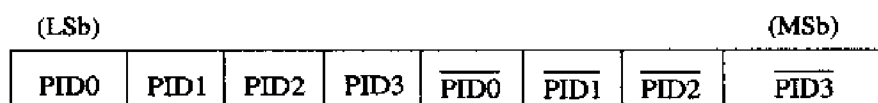


图 10-4 PID 格式

表 10-1 列出了各包标识符的类型、编码及相关说明。

表 10-1 PID 类型

PID 类型	PID 名称	PID[3:0]	说　明
令牌（Token）	OUT	0001B	输出地址＋端点号
	IN	1001B	输入地址＋端点号
	SOF	0101B	帧开始标记与帧号
	SETUP	1101B	控制管道建立地址＋端点号
数据（DATA）	DATA0	0011B	偶数据包
	DATA1	1011B	奇数据包
	DATA2	0111B	.微帧数据包（高速设备）
	MDATA	1111B	一般数据包（高速设备）
握手（Handshake）	ACK	0010B	接收器接收到无误数据包
	NAK	1010B	接收设备不能接收数据或发送设备不能发送数据
	STALL	1110B	端点挂起
	NYET	0110B	设备收到无误数据包，但还没准备好下一个
专用（Special）	PRE	1100B	主机发出低速设备通信前导信号
	ERR	1100B	报告分割传输时的错误（高速设备）
	SPLIT	1000B	分割传输令牌包 PID（高速设备）
	PING	0100B	流量控制检测（高速设备）

● 地址字段（Address）

地址字段指明外设地址，如图 10-5 所示。ADDR<6:0>共指定 128 个地址，每个地址对应一个单一外设。设备在复位及加电时，地址默认为零，具体值需由主机在枚举时指定。



外设零地址为默认地址，不可被分配用于其他用途。



图 10-5　地址字段

当外设有多个端点时，由 4 位端点字段进行寻址，如图 10-6 所示。对于低速设备，每个外设最多提供 3 个通道，而全速设备和高速设备可支持最多 16 个任意类型通道。

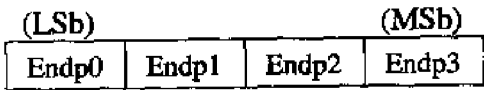


图 10-6　端点字段

- 帧号字段（Frame Number）

帧号字段是一个 11 位长的字段，主机每通过一帧就将其内容加 1，达到最大值 7FFH 时归零。

- 数据字段（Data）

数据字段的长度可在 0~1023 字节间变动，但必须是整数个字节。

- CRC 校验字段

校验字段用来校验所有非 PID 字段。其中，标记字段使用 5 位 CRC 字段，数据字段使用 16 位 CRC 字段。

4. 包格式

USB 传输中的包有 4 种：令牌包、数据包、握手包以及 SOF 包。

- 令牌包（Token Packets）

令牌包由 PID、ADDR、ENDP 和 CRC5 字段构成。其中，PID 字段指明包的类型；而 ADDR 和 ENDP 字段惟一确定接下来将收到数据包的端点；CRC5 字段负责校验 ADDR 和 ENDP 字段。令牌包格式如图 10-7 所示。

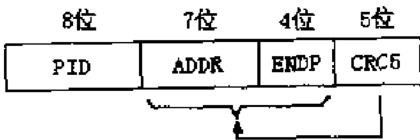


图 10-7　牌包格式

- 数据包（Data Packets）

数据包由 PID、DATA 和 CRC 字段组成。数据字段长度对低速设备而言是 0~8 字节，对全速设备而言是 0~1023 字节，而对高速设备而言则是 0~8192 字节。数据包有 4 种类型，由不同的 PID、DATA0、DATA1、DATA2 以及 MDATA 来区分。数据包格式如图 10-8 所示。



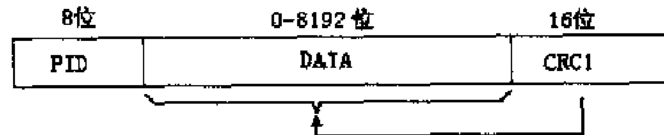


图 10-8　数据包格式

- 握手包（Handshake Packet）

握手包仅由 PID 构成，主要用来报告数据事务的状态。握手包有以下 4 种类型：

  - ACK 包：表示数据包无误接收。
  - NAK 包：用于表示外设暂时不能向主机传输数据，或从主机接收数据。
  - STALL 包：表示外设不能传输或接收数据，或者不支持一个控制管道请求。
  - NYET 包：表示外设接收到了无错误的数据包，但是还没有准备好接收下一个数据包。

- SOF 包（Start Of Frame）

SOF 包是由 PID、帧号字段 Frame Number 及 CRC 字段组成。主机以每 1.00 ms ± 0.0005 ms（低速和全速设备）或是 125us±0.0625us（高速设备）的速率向设备发送 SOF（帧开始）包。它以精确的时间间隔发送 SOF 标识（Marker）及帧数。SOF 包格式如图 10-9 所示。



图 10-9　SOF（帧开始）包

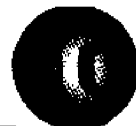
5. 描述符

任何一个 USB 标准设备都包含一个设备描述符表，用于说明设备属性，一般固化在设备内部。当主机检测到总线上有设备插入时，通过控制传输从默认通道中将设备的描述符读出。

USB 设备的描述符有 3 种：设备描述符（Device Descriptors）、设备类描述符（Class Descriptors）和用户自定义描述符（User Specific Descriptors）。其中设备描述符又分为 4 种：配置描述符（Configuration Descriptors）、接口描述符（Interface Descriptors）、端点描述符（Endpoint Descriptors）和字符串描述符（String Descriptors），它们之间的关系如图 10-10 所示。

10.2.3　USB 设备请求

前面提到过：USB 设备的某些属性要由主机通过默认控制通道进行设置或读取。这些操作是通过主机向设备发送 SETUP 包来完成的。每个 SETUP 包有 8 个字节，分为 5 个域，



如图 10-11 所示。图 10-11 中只标明了 bmRequestType 域的组成及含义，SETUP 包中其余的内容包括一个 bRequest 值（具体请求如表 10-2 所示）、一个 wValue 代码（其含义与具体的请求相关）、一个 wIndex 值（当控制请求寻址端点或接口时，wIndex 值指出具体地址）、一个 wLength 域（指出控制事务的数据阶段要传输的数据量，为 0 表示该事务没有数据段）。

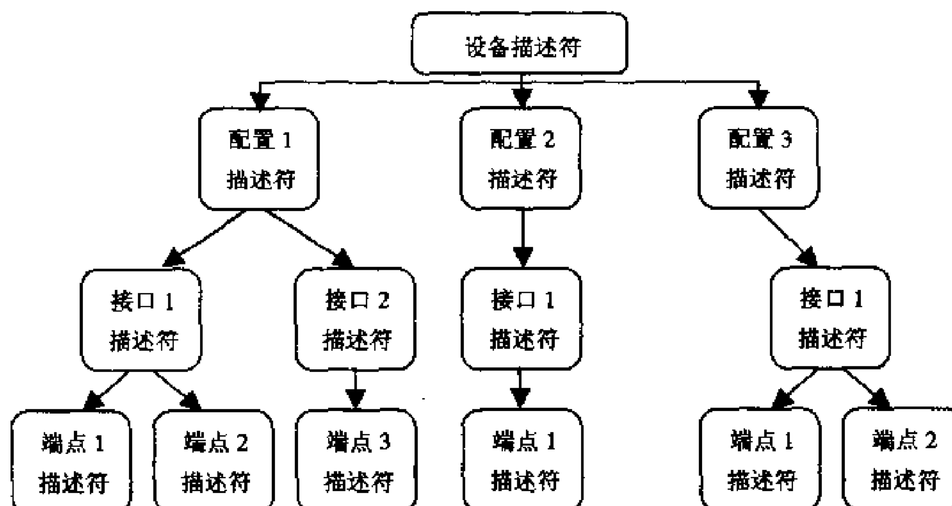


图 10-10 设备描述符关系示意图

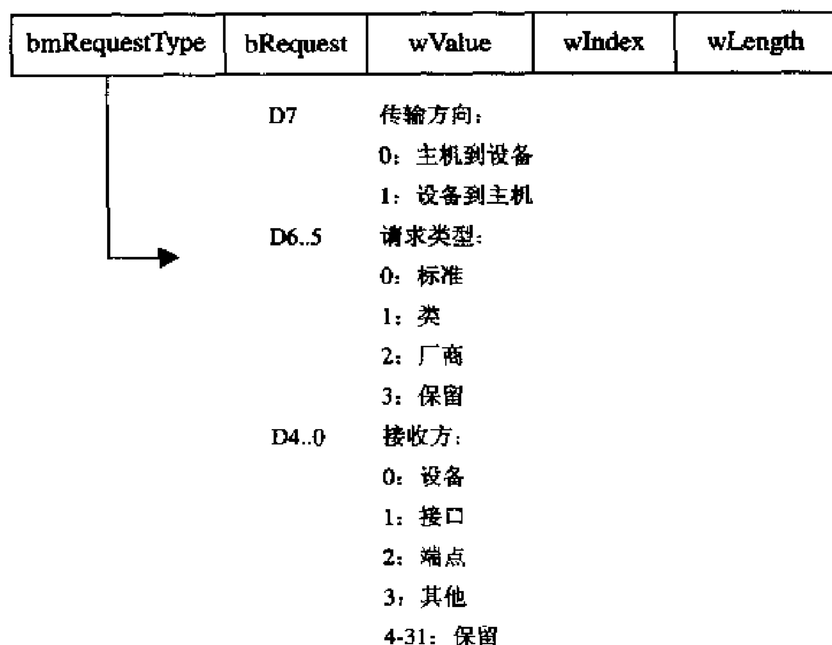


图 10-11 SETUP 数据包格式

USB 规范定义了一系列所有设备必须支持的标准请求，如表 10-2 所示。另外，一个设备类可定义更多的请求，设备厂商也可定义设备支持的请求。



表 10-2　标准设备请求

请求代码	符号名	描述	可能的接收者
0	GET_STATUS	获得状态信息	任何
1	CLEAR_FEATURE	清除一个双态特征	任何
2		（保留）	
3	SET_FEATURE	设置一个双态特征	任何
4		（保留）	
5	SET_ADDRESS	设置设备地址	设备
6	GET_DESCRIPTOR	取设备、配置，或串描述符	设备
7	SET_DESCRIPTOR	设置一个描述符（可选）	设备
8	GET_CONFIGURATION	取当前配置索引	设备
9	SET_CONFIGURATION	设置一个新的当前配置	设备
10	GET_INTERFACE	取当前的 alt 接口索引	接口
11	SET_INTERFACE	使能 alt 接口设置	接口
12	SYNCH_FRAME	报告同步帧号	（等时）端点

10.2.4　USB 设备枚举

当 USB 设备连上主机或是从主机移走时，主机启动一个总线枚举进程，以识别和管理设备的状态变化。其总线枚举的过程如下：

（1）USB 设备连上主机，所连的集线器通知主机本设备已连接上。此时，USB 设备处于加电状态，它所连接的端口是无效的。

（2）主机一旦得知新设备已连上它的某个端口，它至少要等待 100ms 以确保插入操作的完成以及设备电源稳定工作。然后，主机向该端口发出端口使能及复位命令。

（3）集线器将发向端口的复位信号持续 10ms，当复位信号撤消后，端口已经变为有效。这时 USB 设备处于默认状态，并且可以从主机汲取小于 100mA 的电能，所有设备寄存器及状态已经被复位，设备可对默认地址（00H）产生响应。

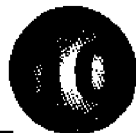
（4）主机给设备分配一个惟一的地址，设备以后就只对该地址进行响应。

（5）主机读取 USB 设备描述符，确认 USB 设备的属性。

（6）主机依照读取的 USB 设备描述符来进行配置，如果设备所需的 USB 资源得以满足，主机就发送配置命令给 USB 设备，表示配置完毕。从设备的角度来讲，它已经准备就绪了。

（7）为了节省电源，当总线保持空闲状态超过 3ms 以后，设备驱动程序进入挂起状态。挂起状态时设备的消耗电流不超过 500uA。当被挂起时，USB 设备保留了包括其地址和配置信息在内的所有内部状态。

当 UBS 设备被取走时，集线器同样会通知主机，主机使对应端口无效，并更新拓扑信息。



10.2.5 小结

USB 设备简单易用，但其实现协议却是非常复杂的，限于篇幅，本章不可能给读者进行面面俱到的介绍。这里的内容只是让读者对 USB 系统及协议有一个整体了解，以便更好地理解下面即将讨论的 USB 设备驱动程序设计的内容。如果读者需要详细了解 USB 规范，可以访问 http://www.usb.org，从中可以找到所有关于 USB 的内容。

10.3 USB 设备驱动程序设计

有了前面介绍的 USB 总线的基础知识后，就可以开始了解 USB 设备驱动程序的细节内容了。

10.3.1 USB 设备驱动程序分类

一个 USB 系统由主机和设备共同组成。所以，对驱动程序而言，它也分为主机端设备驱动程序（Device Driver）、主机控制器驱动程序（Host Controller Driver）和设备端驱动程序（Slave Device Driver）。

1. 主机端设备驱动程序

这部分驱动程序就是常说的设备驱动程序，是在主机环境中为用户应用程序提供一个访问 USB 外设的接口。无论是 Linux 还是 Windows 操作系统，都已经为这部分驱动程序提供了编程接口，驱动程序设计者只需按要求建立程序框架，通过调用操作系统提供的 API 接口函数即可完成对 USB 外设的特定访问。10.3.2 节将对这类设备驱动程序的编写作简要介绍。

2. 主机控制器驱动程序

这部分驱动程序主要是对 USB 主机控制器的驱动，普通用户通常对此比较陌生。这是因为在大多数计算机环境特别是 PC 环境下，主机控制器驱动都是由操作系统提供的。嵌入式设备一般都没有 USB 主机控制器，而只是工作在 Slave 模式下。

如果想让设备具有 USB 主机功能，那么设备中就需要选用一个带有主机控制器的 USB 接口控制芯片，同时还有自己实现该主机控制器的驱动程序。这样的芯片目前市场上并不多，主要有 Scanlogic 公司的 SL811HS/T、Philips 公司的 ISP1161 等。现在，USB 规范又推出了 USB OTG（On-The-Go），它是对 USB 2.0 协议的一个补充。符合该协议的接口芯片具有双重功能：带有该芯片的设备既可以用作主机，也可以用作外设，其主机处理协议（HNP）用于转换 USB 主机和外设功能。这样的芯片常用的有 Philips 公司的 ISP1362，它与 ISP1161



的区别在于 ISP1362 使用同一个接口就可以实现 USB 主机和外设功能，而 ISP1161 是通过不同的接口来分别实现的。

目前，Linux 内核中只提供对 USB 主机控制器的开放主机控制器接口（OHCI）和通用主机控制器接口（UHCI）两种规格的支持，而这两种规格主要被应用在 PC 架构中。uClinux 2.4.x 版本源代码的 src\driver\usb 目录中有一个 hc_sl811.c 文件，可以作为支持 SL811HS/T 接口芯片的主机驱动程序设计的范例参考（基于本书定位与篇幅，就不对它进行分析了，感兴趣的读者可以自己去阅读其源代码）。

USB 主机端设备驱动程序与主机控制器驱动程序的关系如图 10-12 所示，其中的 USB 核是 Linux 的一个子模块，集中定义了一组 USB 相关的数据结构、宏及 API 函数。

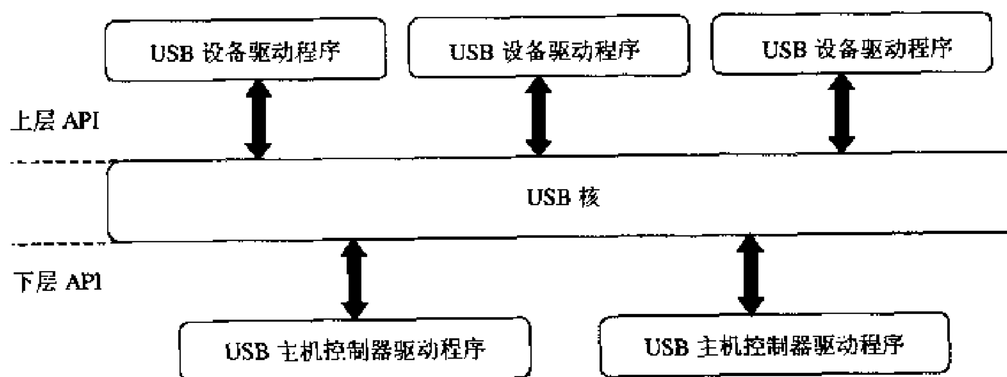


图 10-12　USB 主机端驱动程序结构

3. 设备端驱动程序

USB 设备端驱动程序是常说的设备固件程序（Firmware）的一部分，提供设备信息与主机的通信接口，这是本章后面要重点讨论的内容。设备端的 USB 接口芯片工作在 Slave 模式下，这样的接口芯片很多，各大芯片厂商都有提供，有些厂商还将它嵌入了 CPU 中。但有一点无疑是相同的，就是这些接口都必须符合 USB 规范，因此驱动程序的处理对象及处理方法也基本相同。

10.3.2　主机端设备驱动程序分析

要在主机中为 USB 外设写一个驱动程序并不是一件很困难的事情，当然，前提是必须熟悉 USB 协议规范。编写自己的设备驱动程序最方便快捷的方式是：在 Linux 源代码中找一个与设备类似的驱动程序例子，然后根据自己设备的要求在上面作一些修改，而后编译入内核或作为模块加载即可。

事实上，有关 USB 总线的通用操作如 USB 设备列举、具体数据传输等已经由主机控制器驱动程序完成了，所以这部分驱动程序只需完成与具体设备相关的操作即可。

下面以一个 USB 通用框架驱动程序为例来分析主机端设备驱动程序的结构及编程方法，程序源代码为 src\driver\usb\usb_skelton.c（在分析过程中，需要读者对 USB 之外的 Linux

基础知识比较了解)。

1. USB 设备驱动程序框架

USB 设备驱动程序第一步要做的事情是向 Linux 内核注册它自己，并告诉系统它所支持的设备类型以及它所支持的操作。所有这些信息是通过一个 usb_driver 结构来传递的。一个典型的 usb_driver 结构如下：

```
static struct usb_driver skel_driver = {
    name:       "skeleton",
    probe:      skel _probe,
    disconnect:skel _disconnect,
    fops:       & skel _fops,
    minor:      USB_ SKEL_MINOR_BASE,
    id_table:  skel _table,
};
```

其中，name 是驱动程序的名称；probe 及 disconnect 是两个函数指针，probe()例程在设备列举时被调用，而 disconnect()例程在设备卸载时被调用；fops 是可选的，它指向一个 file_operations 结构，内核通过它来访问驱动程序的文件操作函数，与用户程序的 read、write 等操作进行交互。当然，如果设备是一个 USB 鼠标，就用不到这个结构了。根据设备实际需要，可以定义 file_operations 结构如下：

```
static struct file_operations skel _fops = {
    owner:      THIS_MODULE,
    read:            skel _read,
    write:skel _write,
    ioctl:           skel _ioctl,
    open:skel _open,
    release:    skel _release,
};
```

这样定义后，当用户应用程序执行如 read、write 等操作时，对应的驱动程序例程 my_read()、my_write()将被调用。

Minor 指向设备的次设备号，用于系统识别主设备号相同的设备（即一个驱动程序可同时支持多个相同的 USB 设备）。

Id_table 结构中保存设备的厂商 ID 和产品 ID，作为该设备的惟一标识。驱动程序向系统注册后，当下次插入设备时，系统就可以根据这个标识查找到正确的驱动程序，实现设备的即插即用。

还有一个重要的数据结构是 usb_device 结构，驱动程序在设备探测阶段需要通过它获



得设备的相关配置信息。usb_device 结构在 include\linux\usb.h 中定义：

```
struct usb_device {
...
    struct usb_device_descriptor descriptor; /* Descriptor */
    struct usb_config_descriptor *config;    /* All of the configs */
    struct usb_config_descriptor *actconfig;/* the active configuration */
...
};
```

编写设备驱动程序的主要工作就是实现 usb_driver 结构中的 probe()函数和 disconnect()函数，以及 file_operations 结构中的 read()、write()、open()等函数。下面几个步骤分别讨论这些函数。

（1）USB 驱动程序的注册与注销。

驱动程序的注册很简单，只需在初始化函数中将填好的 usb_driver 结构作为参数传递给 usb_register()函数即可。函数调用方法如下：

```
result = usb_register(&skel_driver);
```

需要说明的是，由于 USB 设备已经有了一个静态分配的主设备号，所以它的注册函数中就不需要主设备号参数了。

驱动程序的注销也同样简单，也是将 usb_driver 结构作为参数传递给 usb_deregister()函数。注销函数的调用方法如下：

```
usb_deregister(&skel_driver);
```

（2）probe()函数。

当一个设备插入 USB 总线时，主机对该设备进行总线枚举，完成对设备的配置并读入设备属性。相关设备属性如设备描述符、配置描述符等信息被主机控制器驱动程序保存在一个 usb_device 结构中。而后，内核根据设备标识找到对应的驱动程序，并调用该驱动程序的 probe()函数，usb_device 结构被作为参数传入该函数。

在 probe()函数中，程序首先确认插入的设备已在内核表中注册过，即比较程序定义的厂商 ID 和产品 ID 是否与通过 usb_device 传入的数值相同，以确保驱动程序的正确性。而后，程序为一自定义的 usb_skel 结构分配内存，并从 usb_device 结构中获得所需信息，最后返回该 usb_skel 结构指针。usb_skel 结构指针定义如下：

```
struct usb_skel {
    struct usb_device *     udev;           /*usb_device 结构指针*/
```

```
    struct usb_interface *  interface;        /*设备接口结构指针*/
    devfs_handle_t          devfs;            /*设备文件节点*/
    unsigned char           minor;            /*次设备号*/
    unsigned char           num_ports;        /*设备端口数*/
    char num_interrupt_in;                    /*端点中断号*/
    char    num_bulk_in;                      /*批量输入端点号*/
    char    num_bulk_out;                     /*批量输出端点号*/

    unsigned char *  bulk_in_buffer;          /*数据接收缓冲区*/
    int          bulk_in_size;                /*接收缓冲区大小*/
    __u8 bulk_in_endpointAddr;                /*批量输入端点地址*/

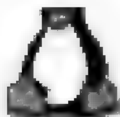
    unsigned char *  bulk_out_buffer;         /*数据发送缓冲区*/
    int              bulk_out_size;           /*发送缓冲区大小*/
    struct urb *     write_urb;               /*数据发送 URB*/
    __u8 bulk_out_endpointAddr;               /*批量输出端点地址*/

    struct tq_struct   tqueue;                /*唤醒任务结构队列*/
    int          open_count;                  /*端口开放次数*/
    struct semaphore sem;                     /*结构内存锁*/
};
```

在这个通用框架驱动程序中，probe()函数需要从设备信息中确认作为批量输入和批量输出的端点号，并为设备传输分配内存，还创建了一个 USB 请求块（URB）以向设备写入数据。在这里，只是看看驱动程序如何在 probe()函数中定位批量输入端点，其他内容读者可以自行阅读源代码 usb_skelton.c 中的对应函数。

```
iface_desc = &interface->altsetting[0];
for (i = 0; i < iface_desc->bNumEndpoints; ++i) {
    endpoint = &iface_desc->endpoint[i];

    if ((endpoint->bEndpointAddress & 0x80) &&
        ((endpoint->bmAttributes & 3) == 0x02)) {
        /*找到批量输入端点*/
        buffer_size = endpoint->wMaxPacketSize;
        dev->bulk_in_size = buffer_size;
        dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
        dev->bulk_in_buffer = kmalloc (buffer_size, GFP_KERNEL);
        if (!dev->bulk_in_buffer) {
            err("Couldn't allocate bulk_in_buffer");
            goto error;
```




```
        }
    }
```

对照 USB 规范的端点描述符结构，可以看到：端点的 bEndpointAddress 字段的 D7 位指示端点方向，值为 1 代表输入端点；端点的 bmAttributes 字段的最低两位指示端点属性，值为 0x02 代表该端点为批量传输端点。

（3）open()、read()、write()函数。

如果用户应用程序需要执行 read、write 或 ioctl 等文件操作，则内核根据驱动程序的注册信息调用 file_operations 结构中定义的对应函数。

open()函数没有什么特殊之处，首先它调用了 MOD_INC_USE_COUNT 宏以增加本驱动模块的引用计数，当该计数数值不为零时，表明该驱动模块处于活动状态，不应被卸载。open()函数还将保存当前设备信息的 usb_skel 结构指针保存在系统的 file 结构的 private_data 字段中，以方便其他函数如 read()、write()等访问。file 结构的 private_data 字段是跨系统调用保存状态信息的一个非常有用的资源，在驱动程序编程中经常使用。

read()函数首先从在 open()函数中保存的 file 结构的 private_data 字段中取得保存当前设备信息的结构指针。这样，它就可以通过这个指针对当前设备进行读操作了。例如，可以像下面的语句这样取得该结构指针：

```
struct usb_skel *dev;
dev = (struct usb_skel *)file->private_data;
```

本例中，read()函数调用 usb_bulk_msg 以通过批量输入端点从设备中读取一定字节数据，若读取成功，则将数据转发到用户空间，以便用户应用程序能够读取。其调用过程如下：

```
int retval = 0;
...
/*用批量输入方式从设备中读取数据*/
retval = usb_bulk_msg (dev->udev,
                usb_rcvbulkpipe (dev->udev,
                dev->bulk_in_endpointAddr),
                dev->bulk_in_buffer, dev->bulk_in_size,
                &count, HZ*10);

/*若读取成功，则将结果数据复制到用户空间*/
if (!retval) {
    if (copy_to_user (buffer, dev->bulk_in_buffer, count))
        retval = -EFAULT;
    else
```

```
        retval = count;
}
```

write()函数的过程与 read()函数相似，只是操作顺序倒过来了。还有一个不同点在于：write()函数使用 USB 请求块（URB）来向设备发送数据。当然，也可以在 read()函数中使用 URB 来读取数据，这也是 USB 规范提倡使用的方法。Write()函数中的主要过程分析如下：

```
/*从用户空间复制数据到 URB 中*/
if (copy_from_user(dev->write_urb->transfer_buffer, buffer,
            bytes_written)) {
    retval = -EFAULT;
    goto exit;
}
/*配置 URB*/
FILL_BULK_URB(dev->write_urb, dev->udev,
            usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
            dev->write_urb->transfer_buffer, bytes_written,
            skel_write_bulk_callback, dev);
/*从批量输出端口发送数据*/
retval = usb_submit_urb(dev->write_urb);
...
```

其中的 FILL_BULK_URB 宏用于配置一个批量传输请求块，也可以用 usb_fill_bulk_urb()函数代替，具体信息参看 uClinux 源代码目录下的\include\linux\usb.h 中的定义源代码。

（4）release()函数。

用户应用程序执行 close 操作时，驱动程序的 release()函数被调用。它的作用与 open()函数相对应：正常情况下，它读取保存于 file 结构的 private_data 字段中的当前设备信息，并将设备引用计数及模块引用计数减 1；若发现此时设备已经被意外拔出，则还要释放相关内存资源。

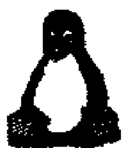
（5）disconnect()函数。

当 USB 设备从系统中拔出，则 disconnect()函数被调用。在这里，所有以前为该 USB 设备分配的数据结构及内存资源均要被释放。

2. 小结

从以上分析看出，一个实际的 USB 设备驱动程序结构并不复杂，有了 Linux 的 USB 核提供的众多数据结构及 API 函数的帮助，驱动程序开发人员可以专注于自身设备的独特应用。

Linux 源代码中提供了许多示例，世界各地的 Linux 爱好者们也在不断公布自己开发的



源代码，这些都是学习 USB 设备驱动程序的有用资源。

10.3.3 设备端 USB 驱动程序分析

1. 设备端 USB 驱动程序结构

这一节来讨论与嵌入式系统结合更紧密的一种 USB 驱动程序：设备端驱动程序。若嵌入式设备只作为 USB 从属设备（Slave Device）使用，则设备中常常没有 USB 主机控制器而只有 USB 接口控制器。驱动程序往往没有内核提供的 API 函数可用，而需自己编程控制数据的组织及收发过程。并且，由于 USB 系统的主从结构，决定了设备方与主机方的行为差异。因此，设备端驱动程序的结构与主机端驱动程序大不相同，同时也更为复杂。

一个完整的设备端驱动程序由以下几部分处理程序组成：

- 初始化例程：完成描述符指针、端点、配置 RAM 等的初始化。
- 数据传输例程：完成控制传输、批量传输、中断传输及同步传输等传输方式下的数据收发工作。
- 标准设备请求处理：处理标准设备请求。
- 厂商请求处理：处理生产商指定请求。
- 其他操作：处理主机发出的端口复位、配置改变等操作。

下面，结合 Motorola 公司的一个 ColdFire 系列的芯片 MCF5272 来看看其 USB 驱动程序的细节内容。Motorola 公司产品网站（http://e-www.motorola.com/）上有该芯片的完整资料，包括下面要分析的范例源代码，若有需要，读者可自己去查阅。

2. 驱动程序接口

USB 驱动程序通过文件系统的设备文件指针被访问，所以，首先需要用 mkmod 命令建立设备文件（本章例子中为 usb0-usb7，分别代表 8 个端点）。其次，内核通过 file_operations 结构来访问驱动程序的函数（这个结构在介绍主机端驱动程序时已经了解了），用户应用程序就可以对驱动程序进行 open、read、write 及 ioctl 等操作。而应用程序要控制 USB 设备驱动程序，如初始化驱动程序或是改变操作模式时，则是通过 ioctl 系统调用来完成的。

典型的 ioctl 命令如下：

- USB_EP_BUSY：查询某个端点是否忙。
- USB_EP_STALL：停止指定端点。
- USB_INIT：调用 usb_init()例程，执行驱动程序的初始化工作。
- USB_GET_FRAME_NUMBER：获取当前帧号。

3. 驱动程序初始化

用户应用程序要使用驱动程序对 USB 设备进行操作之前，必须先发一个 USB_INIT ioctl 命令，以调用驱动程序的初始化例程。本章下面的例子中为 usb_init()例程，在该例程中进行如下对象的初始化工作。

（1）描述符指针及变量的初始化。

初始化指向设备配置 RAM 的指针、设备描述符指针及设备描述符的大小，指针初始化代码如下：

```
pConfigRam = (uint32 *)((uint32)imm + MCF5272_USB_CFG_RAM);
pDevDesc = (uint32 *)usb_get_desc(-1, -1, -1, -1);
DescSize = usb_get_desc_size() + 3;
```

（2）端点初始化。

初始化端点 0 的各个属性，其余端点的属性要由主机设置，端点初始化的操作代码如下：

```
ep[0].ttype = CONTROL;
ep[0].packet_size = ((USB_DEVICE_DESC *)pDevDesc)->bMaxPacketSize0;
ep[0].fifo_length = (uint16)(ep[0].packet_size * FIFO_DEPTH);
ep[0].buffer.start = 0;
ep[0].buffer.free = 0;
```

注意：端点 0 的传输属性必须设置为控制属性。

（3）配置 RAM 初始化。

设备配置 RAM 主要用来保存设备属性，如设备描述符，配置 RAM 初始化代码如下：

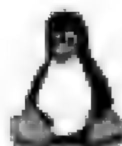
```
MCF5272_WR_USB_EP0CTL(imm, 0);
for (i = 0; i < (DescSize/4); i++)
    pConfigRam[i] = pDevDesc[i];
```

注意：需要注意的是，要对配置 RAM 进行修改，首先要使之无效，这里用一个宏 MCF5272_WR_USB_EP0CTL(imm, 0)来完成，然后才能进行相关赋值操作。

（4）FIFO 缓冲区初始化。

根据 USB 规范，每个端点都有输入和输出两个方向，即 IN 和 OUT（相对于主机而言），所以，每个端点应该设置两个缓冲区，用两个 USB_EP_STATE 结构数组指针来分别指向它们。USB_EP_STATE 结构如下：

```
typedef struct {
    uint32 fifo_length;          /*FIFO 缓冲区大小*/
    uint32 in_fifo_start;        /*IN 缓冲区起始地址*/
    uint32 out_fifo_start;       /*OUT 缓冲区起始地址*/
    uint32 packet_size;          /*包尺寸最大值*/
    uint8 ttype;                 /*数据传输类型*/
    uint8 dir;                   /*传输方向*/
```




```
    USB_BUFFER buffer;          /*数据缓冲区*/
} USB_EP_STATE;
```

其中的 USB_BUFFER 结构定义如下：

```
typedef struct {
    uint8 *start;                    /*缓冲区起始地址*/
    uint16 position;                 /*偏移量指针*/
    uint16 length;                   /*缓冲区长度*/
    uint8 free;                      /*该缓冲区被分配与否*/
} USB_BUFFER;
```

而后将各端点的状态分别保存在两个数组中，其代码如下：

```
USB_EP_STATE *pIN[NUM_ENDPOINTS];
USB_EP_STATE *pOUT[NUM_ENDPOINTS];
uint8 nIN, nOUT, i;
/*端点 0 始终为双向的*/
pIN[0] = &ep[0];
pOUT[0] = &ep[0];
nIN = nOUT = 1;
/*对其他活动端点分类 */
for (i = 1; i < NUM_ENDPOINTS; i++)
{
    if (ep[i].ttype != DISABLED)
    {
        if (ep[i].dir == IN)
            pIN[nIN++] = &ep[i];
        else
            pOUT[nOUT++] = &ep[i];
        }
    }
```

（5）中断初始化。

设置各端点的中断向量及中断级别，具体操作请参看\src\usb\路径下的源代码。

最后，设置相关控制寄存器以使能 USB 控制器、配置 RAM 以及端点 0 的中断，整个初始化工作就完成了。当然，对于其他端点，要等到主机对其进行配置后方可使用。

4. 数据传输

在主机对将要进行操作的端点正确配置后，就可以通过该端点进行数据传输了。以设备向主机方向的数据传输即 IN 传输来看看相关例程编程细节。同步传输方式的数据传输比

较特殊，把它放到本章的最后讨论。而中断传输、控制传输和批量传输这 3 种数据传输方式的情况比较相似，这里先讨论这 3 种传输方式的情况。

当用户应用程序对设备文件调用 write()函数时，usb_device_write()例程被调用，该例程的功能放在 usb_tx_data()例程中完成。其例程定义如下：

```
uint32   usb_tx_data(uint32 epnum, uint8 *start, uint16 length)
usb_tx_data()例程有 3 个参数：
```

- epnum：端点号。
- start：要传输数据的缓冲区指针。
- length：传输字节数。

例程首先判断端点各状态是否正确，如果正确，则设置端点缓冲区属性：

```
ep[epnum].buffer.start = start;          /*缓冲区起始地址*/
ep[epnum].buffer.length = length;        /*缓冲区长度*/
ep[epnum].buffer.position = 0;           /*下一帧数据起始位置，开始为 0 */
```

然后，根据 FIFO 缓冲区的大小，复制合适字节数的数据到 FIFO 缓冲区，并将 buffer.position 设置为已写入的数据大小，函数返回。具体操作流程如图 10-13 所示。

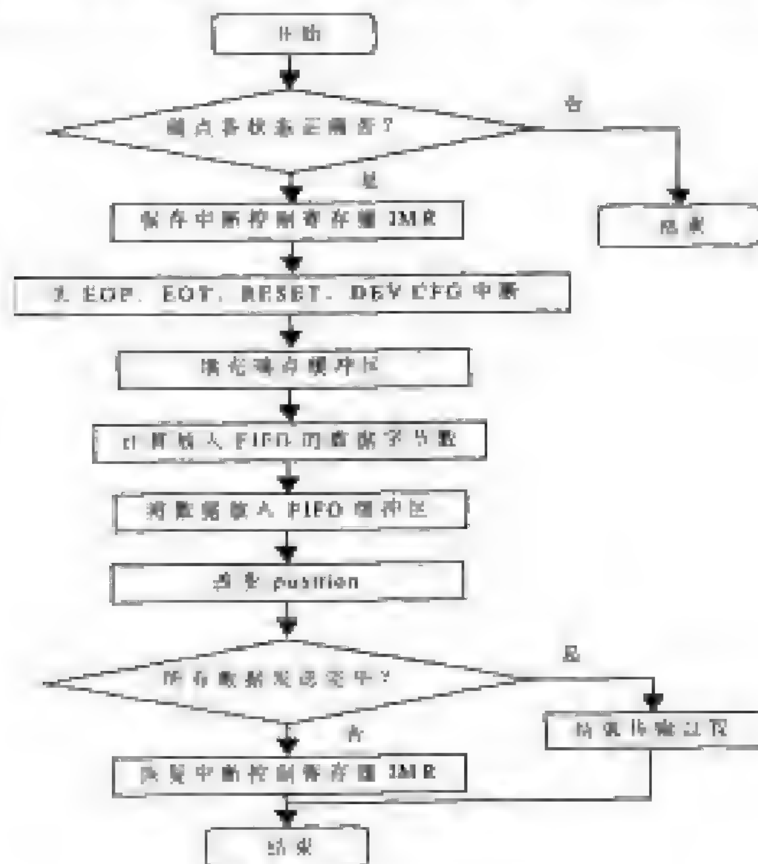


图 10-13　usb_tx_data()函数算法

注意：write()函数是一个异步操作，即不管传输是否完成，用户程序可以接着做其他事情。如果用户想提前发送新的数据，即刷新数据缓冲区，它应该向驱动程序发送 ioctl 命令来查询该端点的状态，以决定是否应该刷新数据缓冲区。

当第一帧数据传输完成时，会触发一个 EOP（End Of Packet）或 EOT（End Of Transfer）中断，调用 usb_in_service()例程，在这个例程中处理剩余数据的发送工作。usb_in_service()例程定义如下：

```
void    usb_in_service(uint32 epnum, uint32 event)
```

usb_in_service()例程有 2 个参数：

- epnum：端点号。
- event：触发本例程的中断类型。

首先判断中断类型是否为 EOP 或 EOT，如果都不是的话，程序直接返回，即不对其他任何中断作出反应。如果是 EOP 中断触发的，则准备下一帧要发送的数据，复制到 FIFO 缓冲区，并根据是否还有未发数据决定是否结束本次传输。如果是 EOT 中断触发的，则直接结束本次传输。具体过程如图 10-14 所示。

图 10-14　usb_in_service()函数算法

从上面的分析过程可以看到：用户程序调用一次 write()函数实际上只负责开始一部分数据的传输，而后面的数据（如果有）则是在 EOP 或 EOT 中断处理程序中完成的。

数据的传输中还有一个关键点，是关于结束传输过程的处理，下面展开讨论。

一次 IN 数据传输由设备端的用户应用程序调用 write()函数开始，驱动程序以最大包方式（填满 FIFO 缓冲区）传送数据给主机，对传输过程结束的处理则依具体情况而定，如表 10-3 所示（具体实现代码限于篇幅不再列出）。

表 10-3　IN 传输结束过程的 3 种不同情况

序　号	情　况	结 束 方 式
1	用户一次传输数据字节数不是 FIFO 缓冲区大小的整数倍	驱动程序清除 EPNCTL[IN_DONE]位以发送数据包，将发生 EOT 中断，驱动在 EOT 中断处理程序中清除端点 IN 缓冲区，并置位 EPNCTL[IN_DONE]
2	用户一次传输数据字节数是 FIFO 缓冲区大小的整数倍，且主机成功收到所有数据	当驱动程序发送完最后一个包后，清除端点 IN 缓冲区
3	用户一次传输数据字节数是 FIFO 缓冲区大小的整数倍，但主机没有发回对收到所有数据的确认	驱动程序再发送一个 0 长度包给主机以指示传输完成，而后清除 EPNCTL[IN_DONE]位，EOT 中断发生，驱动在 EOT 中断处理程序中清除端点 IN 缓冲区，并置位 EPNCTL[IN_DONE]

主机到设备方向的数据传输即 OUT 传输时的情况与设备到主机方向的数据传输即 IN 传输情况相似，这里不再讨论。

下面看看同步数据传输的情况，仍然以 IN 传输为例。

本章开始部分谈过，同步数据传输主要用在音频和视频传输中，数据要求实时，但正确率要求不高，与中断传输、控制传输及批量传输这 3 种传输方式不大相同。同步传输中，驱动程序将 I/O 请求（read()和 write()）放入队列中，以确保数据传输的连续性。用户程序的发送代码如下：

```
write(usb_ep1_file,&buffers[0],5);
write(usb_ep1_file,&buffers[1],5);
/* 等待，直到队列中只剩下一个 write()请求 */
ioctl(usb_ep1_file,USB_EP_WAIT,1);
```

它将一组请求放入队列，而后用 USB_EP_WAIT ioctl 查询端点状态，以控制队列大小。

在驱动程序方面，当收到第 1 个 write()请求后，开始传输过程，然后返回控制。当它收到第 2 个 write()请求且该端口仍为忙状态，则驱动程序将这个请求放入队列。用一个 iso_ep 结构数组来保存各个请求的状态，该结构定义如下：

```
typedef struct {
```




```
    uint8 number_of_requests;          /*请求个数*/
    uint8 first_io_requests;           /*第一个请求的序号*/
    uint8 last_io_requests;            /*最后一个请求的序号*/
    uint8 sent_packet_watch;           /*丢帧计数器*/
    uint8 packet_position;             /*缓冲区内部指针*/
    USB_BUFFER requests_queue;         /*请求缓冲区*/
} iso_ep;
```

同步传输机制的特殊之处在于：当第一次 write 请求的最后一个数据包发往主机后（驱动程序收到主机返回的 EOP 中断信号且 FIFO 缓冲区为空），驱动程序唤醒 usb_ep_wait() ioctl 处理函数，并检查队列中有无请求。如果有，则驱动程序取出下一个请求并再次初始化端点缓冲区 ep[epnum].buffer 及 iso_ep[epnum]，继续传输数据，否则就释放缓冲区。然后，用户应用程序可以利用已经完成传输的第 1 个请求的缓冲区准备新的数据，并放入请求队列。同步数据传输就是这样通过不断取出队列中的 write 请求，使得传输得以不间断地进行，对主机而言，就像一个不间断的数据"流"。

关于同步传输，关键是如何确保传输的实时性以及如何监视主机是否丢帧的问题。

假设设备要发送一个包含 5 个数据包的缓冲区数据给主机，而该缓冲区要在 5ms 之后释放，因为这时 AD 转换器产生了新的数据要放进来。如果主机没有及时发出 IN 令牌包，则驱动程序可能无法及时更新缓冲区。实际中可以这样处理：驱动程序每发送一个数据包，就将指向缓冲区的内部指针 position 向前移动一个数据包位置；而不管主机是否发送了 IN 令牌包，以确保缓冲区能在规定时间内更新，即保证了主机所收到数据的实时性。

驱动程序监视主机在接收数据过程中是否有丢帧，其原理是：由于设备每向主机成功发送一帧数据，则设备方产生一个 EOP（End Of Packet）中断；如果主机方丢帧（即没有向设备发送 IN 令牌包），则 EOP 中断不发生。利用这一特征，可以在驱动程序发出一个数据包时将 iso_ep 结构中的 sent_packet_watch 计数器加 1，而在 usb_in_service()函数中将它清零。这样，只要 EOP 中断正常发生，该计数器的值就不会增加，如果主机丢帧，设备方也可以清楚了解丢帧情况。

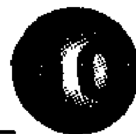
根据以上原理，下面具体分析程序的编写方法。

假设数据缓冲区大小为 5 个数据包大小，即每发生一个 write 请求，设备向主机发送 5 个数据包。根据主机丢帧情况的不同，驱动程序要作不同处理，一般来说有以下 3 种情况：

- 除最后两个包之外的其他某个数据包丢失。

例如第 2 个数据包丢失，如图 10-15 所示。

这种情况下，当 SOF3 中断发生时，程序检测到第 2 个数据包没有发送到主机（即 EOP2 中断没有发生），则驱动程序将 FIFO 缓冲区清空。原则上，这时应该把数据包 3 放入 FIFO。但是，由于从 SOF 中断发生到收到下一个 IN 令牌包的时间间隔实际上小于驱动程序清空 FIFO 并取新数据到 FIFO 的时间，也就是说根本没有足够时间来发送数据包 3。所以，正确做法应该是直接把数据包 4 放入 FIFO 缓冲区，则驱动程序会在第 4 帧中将数据正确送出。



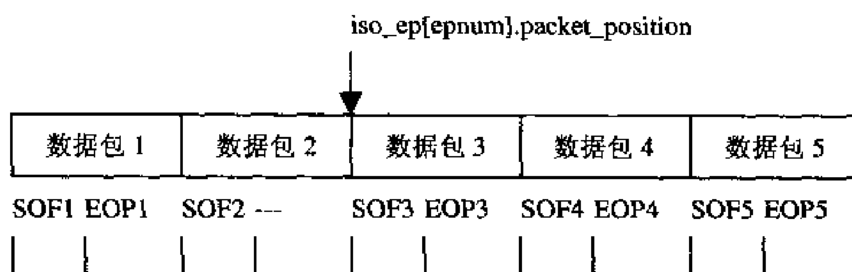


图 10-15　数据包丢失情况 1

- 倒数第 2 个数据包（即数据包 4）丢失

倒数第 2 个数据包（即数据包 4）丢失，如图 10-16 所示。

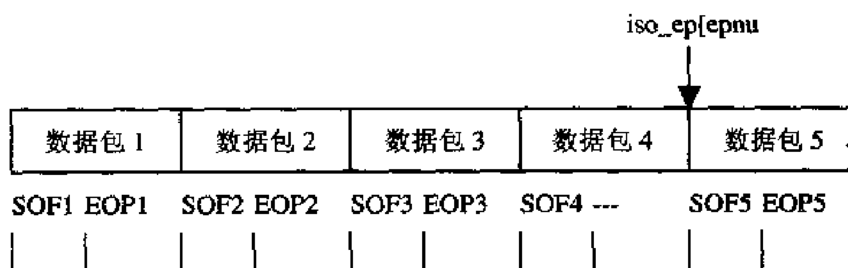


图 10-16　数据包丢失情况 2

如第 1 种（除最后两个包之外的其他某个数据包丢失）情况所述，如果主机丢失数据包 4，则驱动程序也没有足够时间来发送数据包 5，这时在驱动程序中可以直接开始对请求队列中的下一个请求（如果有的话）进行处理。

- 最后一个数据包（即数据包 5）丢失

最后一个数据包（即数据包 5）丢失，如图 10-17 所示。

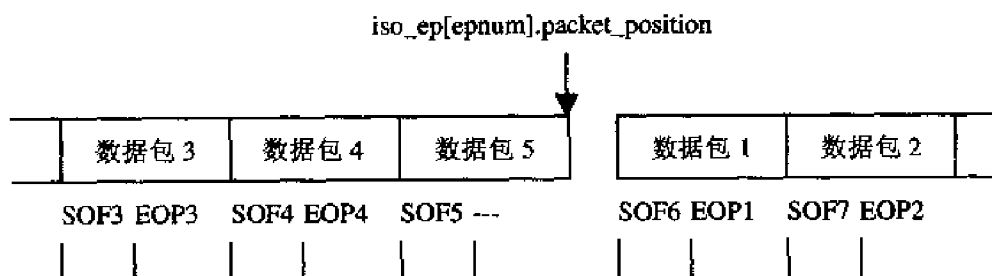


图 10-17　数据包丢失情况 3

这种情况下，驱动程序先判断请求队列是否为空，如果队列为空，则完成此次传输。如果队列非空，则与情况 1（除最后两个包之外的其他某个数据包丢失）类似，应该把新的请求缓冲区中的数据包 2 放入 FIFO 进行发送。

注意：总结以上 3 种情况，可以看到：一旦主机丢失某一个数据包，则它也接收不到



下一个数据包。

5. 厂商请求处理

USB 规范所规定的 11 个标准设备请求中，除 SYNCH FRAME 及 SET DESCRIPTOR 外，其余 9 个都由 MCF5272 内部的 USB 请求处理模块自动处理了，不需驱动程序干预（具体内容请读者参考 MCF5272 芯片资料），SET DESCRIPTOR 请求不做处理。处理器将 SYNCH FRAME 请求作为厂商请求传递给驱动程序，同时要求驱动程序处理的还有 GET DESCRIPTOR 中的字符串描述符获取请求。

驱动程序以默认控制端点（端点 0）响应主机的请求，而主机则是通过发送 SETUP 包来发出请求。硬件检测到主机发出的厂商请求后产生一个 VEND_REQ 中断，驱动程序在该中断处理程序中调用 usb_vendreq_service()函数进行请求处理。示例代码如下：

```
if (event & MCF5272_USB_EP0ISR_VEND_REQ)
{
    /*判断是否为 GET_DESCRIPTOR(String) 请求*/
    if ((MCF5272_RD_USB_DRR1(imm) & 0xFF00 >> 8) == GET_DESCRIPTOR)
    {
        /*如果 STRING Descriptor 存在，则返回该描述符（具体处理代码略）*/
        printf("Host requested the String Descriptor\n");
    }
    /*清除中断标识位*/
    MCF5272_WR_USB_EP0ISR(imm, MCF5272_USB_EP0ISR_VEND_REQ);
    /*调用厂商请求处理函数*/
    usb_vendreq_service(
        (uint8)(MCF5272_RD_USB_DRR1(imm) & 0xFF),
        (uint8)(MCF5272_RD_USB_DRR1(imm) >> 8),
        (uint16)(MCF5272_RD_USB_DRR1(imm) >> 16),
        (uint16)(MCF5272_RD_USB_DRR2(imm) & 0xFFFF),
        (uint16)(MCF5272_RD_USB_DRR2(imm) >> 16));
}
```

其中，请求数据寄存器 DDR1\DDR2 中保存该次请求的类型及各属性值：可从 DDR1 中读取 bmRequestType、bRequest 及 wValue，从 DDR2 中读取 wIndex、wLength，传递给 usb_vendreq_service()函数进行处理。

驱动程序与用户应用程序间的消息传递机制是这样的：驱动程序以 SIGIO 信号通知用户应用程序新的请求的到来；用户应用程序收到该信号后，调用 USB_GET_COMMAND ioctl 命令获取 DEVICE_COMMAND 结构指针。DEVICE_COMMAND 结构定义如下：

```
/* Structure for Request */
typedef struct {
```

```
uint8 bmRequestType;
uint8 bRequest;
uint16 wValue;
uint16 wIndex;
uint16 wLength;
} REQUEST;
typedef struct {
uint8 * cbuffer;                    /*命令缓冲区指针*/
REQUEST request;                    /*主机请求*/
} DEVICE_COMMAND;
```

其中，REQUEST 结构中保存主机请求的各个参数；DEVICE_COMMAND 结构中的 cbuffer 域指向命令缓冲区起始地址，缓冲区长度等于 REQUEST 结构的 wLength 值。显然，cbuffer 域只在主机向设备方传递数据时才有意义；否则，该域无效，不被初始化。

通过以上设定，驱动程序就可以正确响应主机发送的请求了。在此举一个数据 OUT 类型请求的例子来说明。

首先，在驱动程序的初始化阶段需要为 DEVICE_COMMAND 结构开辟一个全局空间，例如下面的代码初始化了一个名为 NewC 的 DEVICE_COMMAND 结构：

```
DEVICE_COMMAND * NewC = NULL;
```

根据 USB 规范，bmRequestType 的第 7 位指示请求的数据传递方向，所以，可以以下方式判断请求类型：

```
if ((bmRequestType < 128) && (wLength > 0))
{
/*这是一个有数据段的请求，且数据传递方向为主机到设备*/
/*为新请求分配内存*/
NewC = (DEVICE_COMMAND *) kmalloc(sizeof(DEVICE_COMMAND) + wLength,
    , GFP_ATOMIC);
/*保存命令缓冲区地址指针*/
NewC -> cbuffer = (uint8 *) NewC + sizeof(DEVICE_COMMAND);
}
```

然后，主机请求命令包含在 SETUP 包的数据段中发送，被驱动程序接收，驱动程序以 SIGIO 信号通知用户应用程序，其代码如下：

```
if ((epnum == 0) && (NewC))
{
/*新命令到达，唤醒 fetch_command_queue()例程*/
```




```
wake_up_interruptible(&fetch_command_queue);
/*并通知用户应用程序*/
if (usb_async_queue)
    kill_fasync(&usb_async_queue, SIGIO, POLL_IN);
}
```

用户应用程序收到该信号后，调用 USB_GET_COMMAND ioctl 命令并从 cbuffer 地址处取得主机命令进行处理。根据它是否能识别该命令，用户程序调用 USB_COMMAND_ACCEPTED 或是 USB_NOT_SUPPORTED_COMMAND ioctl 命令以告诉驱动程序一个它是否识别该主机请求命令的状态标志。驱动程序以该标志为参数调用 usb_vendreq_done()函数，完成这次主机请求响应操作。usb_vendreq_done()函数代码如下：

```
void usb_vendreq_done(uint32 error)
{
    MCF5272_IMM *imm = mcf5272_get_immp();

    if (error)
        /*设备不识别主机请求，返回 STALL 标志并结束本次请求*/
        MCF5272_WR_USB_EP0CTL(imm,
            MCF5272_RD_USB_EP0CTL(imm)
            | MCF5272_USB_EP0CTL_CMD_OVER
            | MCF5272_USB_EP0CTL_CMD_ERR);
    else
        /*正常结束本次请求*/
        MCF5272_WR_USB_EP0CTL(imm,
            MCF5272_RD_USB_EP0CTL(imm)
            | MCF5272_USB_EP0CTL_CMD_OVER);
    return;
}
```

6. 其他操作处理

在设备枚举过程中，主机对设备方的端口复位、改变配置等事件的处理也是通过默认控制端口 0 来完成的。

（1）端口复位事件处理。

当主机发出复位（RESET）命令时，驱动程序在端点 0 的中断处理程序中调用 usb_bus_state_chg_service()函数，以 RESET 值为参数。端口复位主要是清空各端点的端点缓冲区，设置各端点的状态为 USB_DEVICE_RESET，如果全局变量 NewC 指向了一个命令缓冲区的话，也要释放它。示例代码如下：

```
for (i=0; i< NUM_ENDPOINTS; i++)
```

```
{
    ep[i].buffer.start = 0;
    ep[i].buffer.length = 0;
    ep[i].buffer.position = 0;
    ep[i].state = USB_DEVICE_RESET;
}
```

另外，由于复位事件随时可能发生，如果中断处理程序在程序读取某个缓冲区时清空该内存，则会产生严重错误。为避免这种情况的错误，应在操作任何缓冲区之前关闭 RESET 中断，前面数据传输的讨论中可以看到这样的例子。

（2）配置改变事件处理。

驱动程序在端点 0 的中断处理程序中调用 usb_devcfg_service ()函数处理主机的配置改变命令。收到该命令，驱动程序应该清空各端点的端点缓冲区以避免 usb_in_service()和 usb_out_service()例程继续收发数据（其实这也是一种强制中断数据传输的方式），再将各端点状态设置为 USB_CONFIGURED，这样，新的数据传输又被允许。然后，驱动程序根据主机发送到设备的各项配置（保存在 USB 请求处理器的各寄存器中）更新对应全局变量的值。最后，驱动程序以 SIGIO 信号通知用户应用程序所作的配置改变，其示例代码如下：

```
void
usb_devcfg_service(void)
{
static uint8 last_addr = 0;
static uint8 last_config = 0;
static uint32 last_altsetting = 0;
uint8 flags, i, j, epnum;
USB_CONFIG_DESC *pCfgDesc;
USB_INTERFACE_DESC *pIfcDesc;
USB_ENDPOINT_DESC *pEpDesc;
MCF5272_IMM *imm = mcf5272_get_immp();

flags = 0;
if (last_addr != MCF5272_RD_USB_FAR(imm))
{
    #if (DEBUG)
        printf("Address change:   old addr = %d, new addr = %d\n",
                last_addr, MCF5272_RD_USB_FAR(imm));
    #endif
    last_addr = MCF5272_RD_USB_FAR(imm);
    flags |= ADDRESS_CHG;
}
if (last_config != (MCF5272_RD_USB_EP0SR(imm) >> 12))
```




```
{
    #if (DEBUG)
        printf("Configuration change:    old cfg = %d, new cfg = %d\n",
                last_config, MCF5272_RD_USB_EP0SR(imm) >> 12);
    #endif

    last_config = (uint8)(MCF5272_RD_USB_EP0SR(imm) >> 12);
    flags |= CONFIGURATION_CHG;
}
/*其他配置改变处理*/
...   /*略*/
/*通知用户应用程序所作的配置改变*/
usb_devcfg_notice(flags, last_config, last_altsetting);
}
```

10.4 小结

本章介绍 uClinux 下的 USB 设备驱动程序设计。USB 规范在近几年才得到广泛认可和应用，可以说是一门比较新的技术。USB 接口很好用，也很容易用，但正如 USB 协议的设计师们所言："一项技术越是容易使用，它实现起来就越是困难"。USB 协议很复杂，短短十几页的介绍只能是让读者有一个大致的认识，以便在具体的编程过程中有章可循。

在介绍驱动程序设计时，主要分析主机端驱动程序和设备端驱动程序，分别介绍它们的结构以及对 USB 各层协议的具体实现，对其中比较重要的环节给出了代码示例。

驱动程序源代码很长，特别是设备端的程序，因此不可能都在书中列出，读者可以参考 uClinux 的源程序，主机端的程序也可以在 uClinux 源代码目录的\driver\usb\usb_skeleton.c 中找到，设备端驱动程序是基于 Motorola 公司的一块 MCF5272 的评估板，读者可以参看该板子的一些资料以帮助理解。

10.5 思考题

1．USB 总线有哪些数据传输方式？分别适用于哪些场合？

2．USB 通过哪些数据单元来实现控制信息以及数据的传输？

3．说明主机识别 USB 设备的过程。

4．找一个 Linux 不能直接识别的 USB 设备，例如一个 USB 键盘，尝试为它编写主机端设备驱动程序。

5．设备端 USB 驱动程序要完成哪些工作，以便主机能正确识别并操作该设备？

6．列举常用的设备端 USB 接口控制芯片，分析它们的功能，并思考各控制芯片驱动程序的实现有何不同之处。

第 11 章　用 LED 和 LCD 作系统输出

知识点：

- LED 原理与结构
- LED 显示方式
- 把 LED 接入嵌入式系统
- LCD 原理和分类
- 编写 LCD 驱动程序
- LCD 应用实例

本章导读：

在嵌入式系统中，人机接口不但包括可以输入的键盘，还有用于传送信息给用户的输出设备。根据系统应用的需要，输出设备可以是最简单的 LED 指示灯，也可以是功能强大的点阵式 LCD 显示器。本章将介绍如何在嵌入式 Linux 系统中扩展系统输出。

首先，将介绍用 LED 作系统输出，主要包括 LED 发光原理；7 段 LED 显示器结构及 LED 的显示方式等知识，说明如何在嵌入式系统中接入 LED 显示器，如何对它们进行控制。然后，将介绍在嵌入式 Linux 系统中使用 LCD 显示器的相关知识，主要包括 LCD 显示原理、目前使用广泛的 LCD 分类、编写 LCD 驱动程序等。最后以华恒 HHDREZ328-R2 的 LCD 显示模块为例，说明如何在嵌入式 Linux 中开发 LCD 显示器。



11.1 在嵌入式 Linux 系统中扩展 LED 输出

LED 即发光二极管，是嵌入式系统中常用的输出设备。单个 LED 通常用作报警指示、故障指示或提示信号等。而由多段 LED 组成的数码管，则可以显示数字或者少数几个字母，可以用于显示诸如速度、时间等信息。尽管其显示功能有限，但是它成本低，易于扩展，在对输出要求不高的嵌入式系统中是完全可以胜任的。

11.1.1 LED 显示输出的原理和结构

LED 是一个半导体设备。当电流如图 11-1 所示通过它的时候，可以产生可见光。LED 的发光强度与通过 LED 的电流强度成正比。LED 发光的颜色可以是红、黄、绿或者是蓝色。

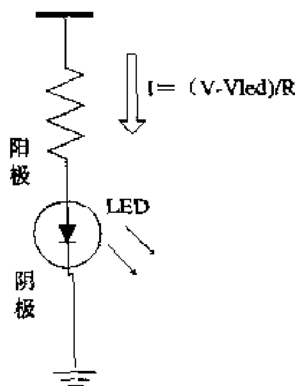


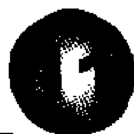
图 11-1 LED 发光原理图

在系统中简单地使用几个 LED，就可以直接与微处理器相连进行控制。微处理器使用一个输出端口，就可以很容易地控制一个或者多个 LED，通过在该端口写入一个 0 电位就可以开启 LED。

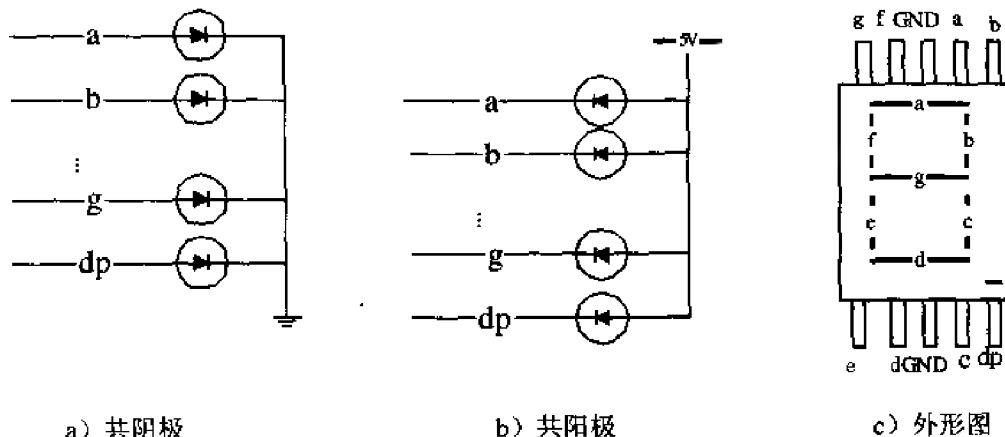
注意：在此，假设该端口可以接收每个 LED 需要的电流。在实际使用过程中，把发光二极管的电流直接输入处理器引脚是不恰当的，应该添加保护电路。

由数个 LED 构成的数码管可以显示数字或者少数几个字母。数码管的构成有 7 段和“米”字型之分，这种显示块有共阴极和共阳极两种。7 段结构的数码管如图 11-2 中所示，共阴极的 LED 数码管的发光二极管阴极连接在一起，通常此公共阴极接地，当某个发光二极管的阳极接高电平时，发光二极管点亮，相应的段被显示。同样，共阳极的 LED 显示块的发光二极管的阳极连在一起，通常此公共阳极接正电压。当某个发光二极管的阴极接低电平时，相应的段被显示。

图 11-2 中有一个 dp 显示段，用于显示小数点。在 LED 数码管中，通过点亮相应段的组合可以构成数字或者字母。在 7 段 LED 数码管中，由于只有 7 个发光段，所以字符码为



一个字节，它的字符码与显示字符之间的关系如表 11-1 所示。



a）共阴极　　b）共阳极　　c）外形图

图 11-2　7 段 LED 结构及外形图

表 11-1　7 段 LED 字符码

显示字符	共阳极字符码	共阴极字符码	显示字符	共阳极字符码	共阴极字符码
0	3FH	C0H	C	39H	C6H
1	06H	F9H	d	5EH	A1H
2	5BH	A4H	E	79H	86H
3	4FH	B0H	F	71H	8EH
4	66H	99H	P	73H	8EH
5	6DH	92H	U	3EH	C1H
6	7DH	82H	I	31H	CEH
7	07H	F8H	Y	6EH	91H
8	7FH	80H	H	76H	89H
9	6FH	90H	L	38H	C7H
A	77H	88H	“灭”	00H	FFH
B	7CH	83H	……	……	……

11.1.2 LED 显示方式

由 N 个 LED 显示块可以构成 N 位的 LED 显示器。图 11-3 是 4 位 LED 显示器的结构原理图。

N 位 LED 显示器有 N 根位选择线和 8×N 根段选择线。根据显示方式的不同，位选择线和段选择线的连接方式也有所不同。其中段选择线控制要显示什么字符，而位选择线则控制显示哪几个数码管。

LED 显示器有静态显示和动态显示之分。下面将分别对这两种显示方式加以介绍。

● LED 静态显示方式

LED 显示器工作于静态显示方式时，各位的共阴极（或者共阳极）连接在一起并接地（若



是共阳极接+5V 电压)，每位的段选择线（a~dp）分别与一个 8 位的锁存输出相连。之所以称为静态显示，是由于显示器的各位互相独立，而且只要显示的字符一经确定，相应锁存器的输出将维持不变，直到显示另一字符为止。也正因为如此，静态显示的亮度比较高。

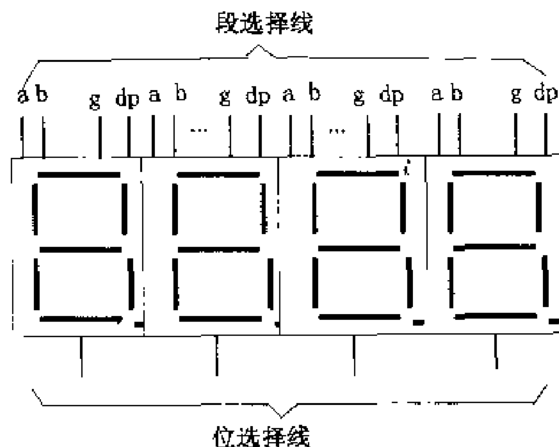


图 11-3　4 位 LED 显示器的构成

若由图 11-3 所示的 4 个数码管构成的 4 位 LED 显示器采用静态显示方式显示的话，只需要把它们的位选择线全部接地（共阴极）。然后，把各个数码管的段选择线与微处理器的 I/O 端口或者锁存器相连即可。这样各位就可以独立显示，只要在该位的段选择线上输出相应的字符码，该位就可以保持相应的显示字符。由于显示器各位的段选择线都是由一个 8 位的端口控制的，所以在同一时间里，各位显示的字符就可以不同。

这种显示方式编程容易，管理也很简单，但它的代价是占用端口资源较多。如果用 I/O 口进行控制，则需要 4 个 8 位 I/O 口。如果用锁存器（如 74LS3773）与之接口，则需要 4 片 74LS373 芯片。如果是“米”字段的数码管，则需要占用更多的端口。在需要显示位数较多的情况下，这种方式是不适宜的，这时应该采用动态显示的方式。

- LED 动态显示方式

在多位 LED 显示时，为了简化硬件电路，通常将所有位段选择线公用，把它们相应地并联在一起，由一个 8 位（7 段 LED）或 16 位（“米”字段 LED）I/O 端口控制，形成段选择线的多路复用。而各位的共阳极或者共阴极分别由相应的 I/O 口控制，实现各位的分时选通。图 11-4 所示为一个 4 位 7 段 LED 动态显示器原理图。其中段选择线占用 8 位 I/O 端口，而位选择线占用 4 位。由于各位的段选择线并联，段选择线的输出对各位来说都是相同的。因此，同一时刻，如果显示器的位选择线都处于选通状态的话，4 位 LED 将显示相同的字符。

如要使人眼看起来各位 LED 显示的是不同的字符，就必须采取扫描显示的方式，即在某一时刻，只让某一位的位选择线处于选通状态，而其他位选择线处于关闭状态。同时，段选择线上必须输出与这一位 LED 相应的字符码。这样，在任何一个时刻，4 位 LED 只有选通的那一位显示出字符，而其他 3 位是熄灭的。然后在下一时刻，选通下一位 LED，并在段选择线上输出下一位的字符码，以显示下一位字符。这样周而复始地循环下去，就可



以使各位LED显示相应的字符，虽然这些字符不是在同一时刻显示的，但是由于人眼有视觉暂留现象，只要它们的显示时间间隔足够小，人眼将很难察觉，这就会造成多位LED同时点亮的视觉效果。

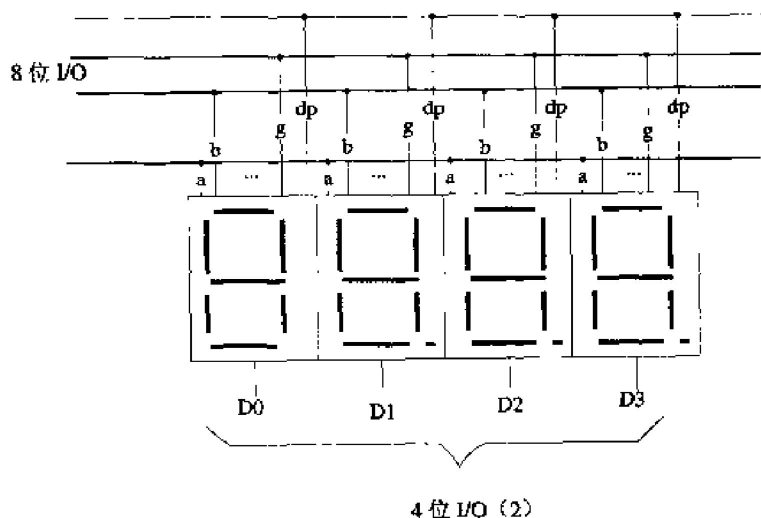
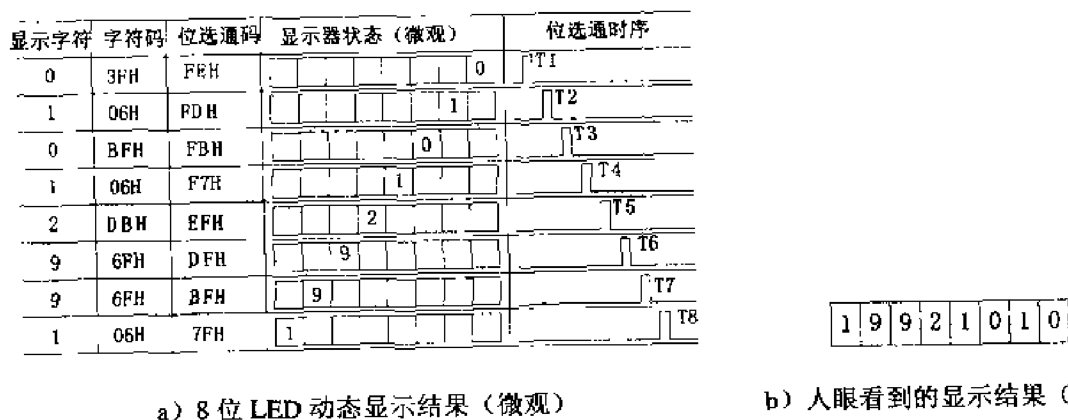


图11-4　4位7段LED动态显示电路

关于动态显示的时间间隔，可以通过定时中断完成。如对于8位LED显示器，扫描显示频率为50 Hz，扫描间隔为1s/50=20ms。假如显示一位保持1ms的时间，则显示完所有8位后，耗时8ms，剩下的12毫秒CPU可以进行其他处理。上述的1ms的定时时间应根据实际情况确定，与采用的数码管型号也有关系。不能太小，因为发光二极管的导通到发光需要一段时间，当导通时间太短，发光太弱人眼无法看清。但也不能太大，因为毕竟需要受限于临界闪烁频率，而且此时间越大，其占用CPU的时间就越多。动态显示的示意图如图11-5所示。



显示字符	字符码	位选通码
0	3FH	FEH
1	06H	FDH
0	BFH	FBH
1	06H	F7H
2	DBH	EFH
9	6FH	DFH
9	6FH	BFH
1	06H	7FH

a）8位LED动态显示结果（微观）　　b）人眼看到的显示结果（宏观）

图11-5　位LED动态显示过程示意图

从动态显示的原理可以看出，它对I/O端口的节省实际上是以牺牲CPU处理时间换来的。显示位太多的话，将耗用CPU的大量处理时间。



11.1.3 在嵌入式 Linux 系统中使用 LED 显示器

如果要在嵌入式 Linux 系统中使用 LED 显示器显示字符，必须按上述过程提供段选码（即字符码）和位选码。其中字符码可以使用软件方法得到，也可以使用硬件译码的方式得到。下面就如何用这两种方式把 LED 显示器接入嵌入式 Linux 应用系统进行说明。

1. 软件方式

在软件方式中，微处理器输出端口必须给出要显示字符的字符码，字符与字符码的映射关系可以参看表 11-1。它的硬件连接方式如 11-6 所示。

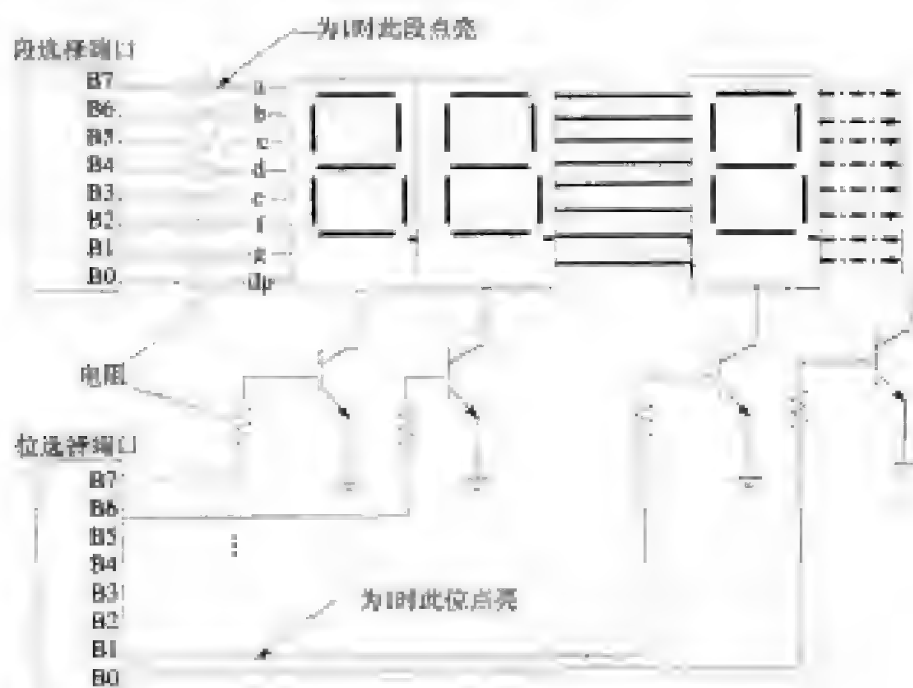


图 11-6 不使用 LED 译码器的连接方式

在这种方式中，可以通过两种方法来得到字符码。一种方法是使用查表的方式，定义一个链表，存储其各个字符和它们对应的字符码。再定义一个查表函数，函数的输入为要输出的字符，函数的输出为这个字符对应的字符码。在这个函数中对链接表进行一次遍历操作就可以得到一个字符对应的字符码了。

如定义如下的数据结构：

```
struct s_charcode{
char ch;
uint8 code;
struct s_charcode * next};
```

这个数据结构用于建立链表，并在其中建立起字符和字符码的对应关系。然后定义查表函数：

```
uint8 get_chcode(char dispch)
{
/*对链接表进行遍历操作，并返回得到的字符码*/
}
```

例如，要在第一位显示数字“1”，可以使用如下方式：

```
outb（0x01,get_chcode("1")）；
```

另一种方法是使用宏定义，即在最开始时定义各个字符的字符码的宏定义，如：

```
#define DISPCH_0 0x3f;
#define DISPCH_1 0x06;
......
```

在输出函数中，使用 outb(0x01,DISPCH_0)就可以显示数字“0”了。

2. 硬件方式

如果使用硬件译码器，如 MC14558，其功能是输入 BCD 码，输出 7 段显示器的字符码，其引脚如图 11-7 所示。其中 A、B、C、D 为 BCD 码输入端，a、b、c、d、e、f、g 为数码管段输出端，En 为使能端，RBI 为消影输入，RBO 为消影输出。

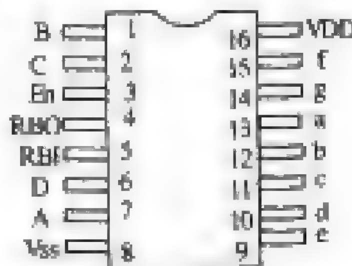


图 11-7 MC14558 引脚图

由于 MC14558LED 译码芯片没有锁存功能，因此常用于动态扫描显示电路中，如果用于静态显示，其前应该加锁存器。由 MC14558 构成的动态刷新的 LED 显示器原理如图 11-8 所示。

在图 11-8 中，CPU 的 p0～p3 口输出 BCD 码，与 MC14558 的 BCD 码输入端即 ABCD 端相连，译码后的字符码经 a、b、c、d、e、f、g 七个引脚输出到 7 段 LED 显示器中，p4～



p6 经过 74LS138 译码器，得到 8 根位选线 Y0～Y7，分别与 8 位 LED 显示器的各位相连，以控制各位数码管的选通。p7 与 3-8 译码器的 S1 端相连，以控制其选通，从而控制整个 8 位显示器是点亮还是熄灭，当 p7 置为高电平时，显示器显示；当 p7 置为低电平时，显示器熄灭。另外，在显示器中还使用了一根 I/O 线来控制小数点的显示，这就是 p8，p8 与各个数码管的 dp 端相连。当某位位选线为低，且此时 p8 口为高时，该位的 dp 段被显示，即小数点被点亮。

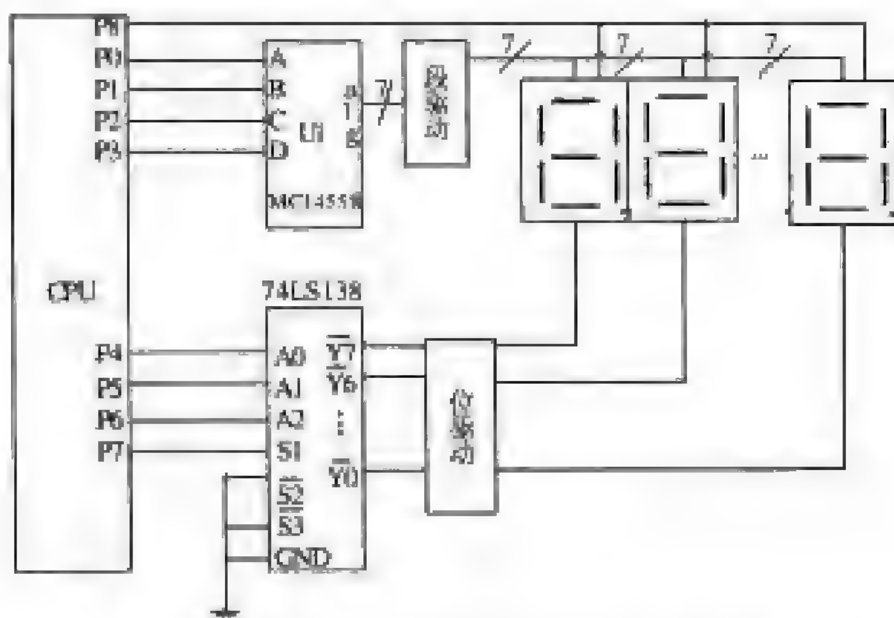


图 11-8　由 MC14558 构成的 8 位动态 LED 显示器

下面将就如何在嵌入式 Linux 系统的软件中，实现对扩展的 LED 显示器进行动态扫描驱动加以说明。假设硬件连线示意图如图 11-8 所示，用 CPU 的低 4 位 p0～p3 作为输出的段选择线；用 p4～p7 作为 3-8 译码器 74LS138 的输入口，为位选择线；用 P8 作为显示小数点的控制端口。如将要显示的日期为 2003.08.25。

注意：在这里直接把 CPU 的输出端口与 LED 显示电路相连，并没有考虑外部端口的寻址。在实际应用中，由于系统通常还有其他外设，而不仅仅是只有 LED 显示器，所以连接情况会有所差别。同时，还需要考虑外部端口的寻址情况。

首先，对显示器进行初始化，熄灭所有的 LED 位，只需关闭 3-8 译码器，使所有的位选择线都为高即可。关闭译码器的方法是把 p7 置为低。

可使用如下代码：

```
outb(0x01,0x00)  //关闭显示
```

注意：如果使用了 BCD 译码器的使能和消影引脚，还应该把它们置为否，以确保消隐显示，在此不考虑它们。

然后，进入显示模块，根据前面介绍的动态显示的原理，需要用到两个定时器，一个用于控制扫描频率，另一个用于显示延时。在这里扫描频率定为 50Hz，即每 20ms 所有 8 位刷新显示一遍，而显示延时定为 1ms，显示子程序如下：

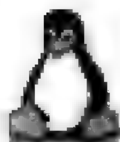
```
struct time_list ledtimer; /*先定义一个长定时器数据结构并对数据结构进行初始化*/
ledtimer.expires=2;/*在许多结构中 HZ＝100，jiffies 每增加 1，经过的时间为 10ms，这里的扫描频率为 50hz，时间间隔为 20ms，所以设定 expires 为 2*/
ledtimer.function=leddisp; /*定义定时时间到了后的处理函数为 leddisp()*/
```

最后，在 leddisp(unsigned long d)函数中对 8 位 LED 显示器动态扫描，在每位上停留显示 1ms，依次显示，如果在哪一位上有小数点，则把 p8 置高。如这里要显示 2003.08.25，第 4 位与第 6 位的小数点都需要置为亮。另外，定时 1ms 是一个短定时，需要用到 udelay()函数，它的参数是微秒级，所以使用 udelay(1000)就可以定时 1ms。这时只需要重新设置好扫描定时器就可以了。例如，要在第一位显示“2”，可以采用下面的代码实现。

```
outw(0x01,0x82); /*显示"2"，其中 p7 置为高是为了使能 3-8 译码器，其中位选线为 000*/
udelay(1000);//在这一位上停留 1ms，以增强发光亮度
```

整个示例程序的源代码如下（见光盘/demo/chap11/11-1）：

```
/*驱动 LED 的示范代码
这段代码只是示范性的，实际使用时应根据开发硬件作相应修改
*/
#include<stdio.h>
#include<signal.h>
#include <asm/param.h>
#include<linux/timer.h>
#include<asm/delay.h>
#include<asm/io.h>
void leddisp(unsigned long d);
struct time_list ledtimer;            //先定义一个长定时器数据结构
ledtimer.expires=2;/*在许多结构中 Hz＝100，jiffies 每增加 1，经过的时间为 10ms，这里的扫描频率为 50Hz，时间间隔为 20ms，所以定 expires 为 2*/
ledtimer.function=leddisp;            //定义定时时间到了后的处理函数为 leddisp()
void init_led()
{
        outb(0x01,0x00)             //关闭显示
        add_timer($ledtimer);       //设置好定时器
}
```




```
/*定时器处理函数*/
void leddisp(unsigned long d)
{
    /*显示“2”，其中 p7 置为高是为了启动 3-8 译码器。因为是第一位，位选线为
  000*/
    outw(0x01,0x82);
    udelay(1000);              //在这一位上停留 1ms，以增强发光亮度
    outw(0x01,0x90);           //显示“0”，位选线为 001
    udelay(1000);
    outw(0x01,0xa0);           //显示“0”，位选线为 010
    udelay(1000);
    outw(0x01,0x1b3);          //显示“3”，位选线为 011
    udelay(1000);
    outw(0x01,0xc0);           //显示“0”，位选线为 100
    udelay(1000);
    outw(0x01,0x1d8);          //显示“8”，位选线为 101
    udelay(1000);
    outw(0x01,0xe2);           //显示“2”，位选线为 110
    udelay(1000);
    outw(0x01,0xf5);           //显示“5”，位选线为 111
    udelay(1000);
    outb(0x01,0x00)            //一轮显示已经完了，清零输出
    add_timer(&timer_list);    //重新设置定时器
}
```

11.2 LCD 显示器的使用

随着嵌入式系统的应用越来越广泛，功能也越来越强大，对系统中的人机界面的要求也越来越高。LCD 液晶显示模块的出现满足了嵌入式系统日益提高的显示要求。它可以根据用户需要显示汉字和字符，也可以显示基于点阵的图形，同时还具有低压、微功耗等优点，因而被广泛应用于嵌入式系统中。

11.2.1 LCD 简介

液晶显示器（Liquid Crystal Display，LCD）具有显示信息丰富、功耗低、体积小、重量轻、超薄等许多其他显示器无法比拟的优点，近几年来被广泛应用于由单片机控制的智能仪器、仪表和低功耗电子产品中。

1. 显示原理

顾名思义，液晶显示器就是采用“液晶”（Liquid Crystal）材料制作的显示器。那什么



是液晶呢？其实，液晶是一种介于固态和液态之间的物质，当被加热时，它会呈现透明的液态，而冷却的时候又会结晶成混乱的固态。液晶是具有规则性分子排列的有机化合物。液晶按照分子结构排列的不同分为 3 种：类似粘土状的 Smectic 液晶、类似细火柴棒的 Nematic 液晶、类似胆固醇状的 Cholestic 液晶。这 3 种液晶的物理特性都不尽相同，用于制作液晶显示器的是第二类的 Nematic 液晶，分子都是长棒状的，在自然状态下，这些长棒状的分子的长轴大致平行。随着研究的深入，人们开始掌握液晶的许多其他性质：当向液晶通电时，液晶分子排列得井然有序，可以使光线容易通过；而不通电时，液晶分子排列混乱，阻止光线通过。通电与不通电就可以让液晶像闸门一样阻隔或让光线穿过。可以控制光线的两种状态是液晶显示器形成图像的前提条件，当然，还需要配合一定的结构才可以实现光线向图像转换。

液晶显示器的显像原理，是将液晶置于两片导电玻璃之间，靠两个电极间电场的驱动，引起液晶分子扭曲向列的电场效应，以控制光源透射或遮蔽功能，在电源开/关之间产生明暗而将影像显示出来。若加上彩色滤光片，则可显示彩色影像。在两片玻璃基板上装有配向膜，所以液晶会沿着沟槽配向，由于玻璃基板配向膜沟槽偏离 90°，所以液晶分子成为扭转型，当玻璃基板没有加入电场时，光线透过偏光板跟着液晶作 90° 扭转，通过下方偏光板，液晶面板显示白色；当玻璃基板加入电场时，液晶分子产生配列变化，光线通过液晶分子空隙维持原方向，被下方偏光板遮蔽，光线被吸收无法透出，液晶面板显示黑色。液晶显示器便是根据此电压有无，使面板达到显示效果。

看到这里，可能读者会问，为什么要把加电时设置为不透光呢？因为在通常状态下显示器都是亮着的，所以设置加电时不透光将更节约能源。

2. LCD 显示器的分类

按显示功能的强弱分，LCD 可分为段位式 LCD、字符式 LCD 和点阵式 LCD。其中，段位式 LCD 和字符式 LCD 只能用于字符和数字的简单显示，不能满足图形、曲线和汉字显示的要求。而点阵式 LCD 不仅可以显示字符、数字，还可以显示各种图形、曲线及汉字，并且可以实现屏幕上下左右滚动、动画、分区开窗口、反转、闪烁等功能，用途十分广泛。

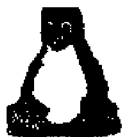
按照液晶显示器的使用场合，其采用的显示模块还可以分为数显液晶模块、点阵式液晶字符模块、点阵图形液晶模块。下面将分别对这 3 类显示模块加以介绍：

（1）数显液晶模块。

数显液晶模块是一种由段式液晶显示器件与专用的集成电路组装成一体的功能部件，只能显示数字和一些标识符号。段式液晶显示器件大多应用在便携、袖珍设备上。由于这类设备体积小，所以尽可能不将显示部分设计成单独的部件，即使一些应用领域需要单独的显示组件，也应该使其除具有显示功能外，还应具有一些信息接收、处理、存储传递等功能。由于它们具有某种通用的、特定的功能，而广受市场的欢迎。常见的数显液晶显示模块应具有以下几种用途：

- 计数

计数模块是一种由不同位数的七段型液晶显示器件与译码驱动器，或再加上计数器装配成的计数显示部件，具有记录、处理、显示数字的功能。目前我国市场上能够见到的主



要产品有由 CD4055 译码驱动器驱动的单位液晶显示器件显示模块，以及由 ICM7211、ICM7231、ICM7232、CD14543、UPD145001、HD44100 等集成电路与相应配套的液晶显示器件组装成的 4 位、6 位、8 位、10 位、12 位、16 位计数模块。

- 计量

计量模块是一种由多位段型液晶显示器件和具有译码、驱动、计数、A/D 转换功能的集成电路片组装而成的模块。由于所用的集成电路中都具有 A/D 转换功能，所以可以将输入的模拟量电信号转换成数字量显示出来。大家知道，任何物理量，甚至化学量（如酸碱度等）都可以转换为模拟电量，所以只要配上一定的传感器，这种模块就可以实现任何量值的计量和显示，使用起来十分方便。计量模块所用的集成电路型号主要有 ICL7106、ICL7116、ICL7126、ICL7136、ICL7135、ICL7129 等，这些集成电路的功能、特性决定了计量模块的功能和特性。作为计量产品，按规定必须进行计量鉴定，经计量部门批准在产品上贴有计量合格证。

- 计时

计时模块是液晶显示器件使用历史最久、应用最广泛的一种应用。由一个液晶显示器件与一块计时集成电路装配在一起就组成了一个功能完整的计时器。由于不少计时模块还具有定时、控制功能，因此这类模块可广泛装配到一些家电设备上，如收录机、CD 机、微波炉、电饭煲等电器上。

（2）点阵式液晶字符模块。

点阵式液晶字符模块是由点阵式字符液晶显示器件和专用的行、列驱动器、控制器及必要的连接器件装配而成的，它可以显示数字和西文字符。这种点阵字符模块有的本身自带字库，有字符发生器，具有显示容量大、功能丰富的特点。一般这种模块最少也可以显示 8 位 1 行或 16 位 1 行以上的字符。这种模块的点阵排列是由 8×8、16×8 或 16×16 的一组一组像素点阵排列组成的。每组的像素组成 1 个字，每个字之间有一定的间隔，每行字之间也都有一行的间隔，所以不能显示图形。

一般在模块控制、驱动器内不仅具有已固化好字符字模的字符库 CGROM，还具有让用户自定义建立专用字符的随机存储器 CGRAM，它允许用户建立自己的点阵字符。

（3）点阵图形液晶模块。

点阵图形液晶模块也是点阵模块的一种，其特点是点阵像素连续排列，行和列在排布中均没有空隔，因此可以显示连续、完整的图形。由于它也是由 X-Y 矩阵像素构成的，所以除显示图形外，也可以显示字符。它又可以分为以下几类：

- 行、列驱动型

行、列驱动型是一种必须外接专用控制器的模块，其模块只装配有通用的行、列驱动器，这种驱动器实际上只有对像素的一般驱动输出端，而输入端一般只有 4 位以下的数据输入端、移位信号输入端、锁存输入端、交流信号输入端等，如 HD44100、HD66100 等。此种模块必须外接控制电路，如 HD61830、SED1330 等才能与计算机连接。该种模块数量最多，应用最普遍。虽然需要采用自配控制器，但它也给客户留下了可以自行选择不同控制器的自由。

- 行、列控制型



行、列控制型是一种可直接与计算机接口，依靠计算机直接控制驱动器的模块。这类模块所用的列驱动器具有 I/O 总线数据接口，可以将模块直接挂在计算机的总线上，省去了专用控制器，因此对整机系统降低成本有很大的好处。对于显示系统的像素数量不大，整机功能不是很复杂的系统非常适用，不过它会占用系统的部分资源。

- 行、列驱动—控制型

行、列驱动—控制型是一种内藏控制器的点阵图形模块，也是比较受欢迎的一种类型。这种模块不仅装有如第一类的行、列驱动器，而且也装配有如 T6963C 等的专用控制器。这种控制器是液晶驱动器与计算机的接口，它以最简单的方式受控于计算机，接收并反馈计算机的各种信息，经过自己独立的信息处理实现对显示缓冲区的管理，并向驱动器提供所需要的各种信号、脉冲，操纵驱动器实现模块的显示功能。这种控制器具有自己的一套专用指令，并具有自己的字符发生器 CGROM，这就要求用户必须熟悉这种控制器的详细说明书，才能进行操作。这种模块使用户摆脱了对控制器的设计、加工、制作等一系列工作，又使计算机避免了对显示器的繁琐控制，节约了主机系统的内部资源。

3. 显示汉字

一般西文为 8×8 点阵，因而显示一个西文字只需要 8 个字节，而每一个汉字要占 4 个西文字体，因此显示一个汉字需要 32 个字节。汉字字库表为一张数据表，每个汉字在数据表中通常由 32 个字节组成一个点阵图形。由于 ASCII 码编码是由 0X00~0X7F 表示，因此，每个汉字可由两个字节 0Xxx 和 0Xyy 来表示，每个字节为 0X80～0XFF（区别于 ASCII 代码）。第一个汉字定义为 0X80 0X80，依此类推，直至 0X80 0XFF、0X81 0X80、…、0XFF 0XFF，总计可以定义 128×128=16384 个汉字。

国家标准信息交换用汉字字符集 GB 2312-80 共收录了汉字、图形符号等共 7445 个，其中汉字 6763 个，按照汉字使用的频度分为两级，其中一级汉字 3755 个，二级汉字 3008 个。汉字、图形符号根据其位置将其分为 94 个“区”，每个区包含 94 个汉字字符，每个汉字字符又称为“位”。其中“区”的序号由 01 区至 94 区，“位”的序号也由 01 位至 94 位。若以横向表示“位”号，纵向表示“区”号，则“区”和“位”构成一个二维坐标。给定一个“区”值和“位”值就可以确定一个惟一的汉字或图形符号，即 4 位阿拉伯数字就可以惟一地确定一个汉字或符号。如“北”字的区位码是“1717”，而“京”字的区位码是“3009”。前两位是区号，后两位是位号。其中 1~15 区是各种图形符号、制表符和一些主要国家的语言字母，16~87 区是汉字，其中 16~55 区是一级汉字，56~87 区是二级汉字。

如果要得到一个汉字的显示点阵，必须给出它在字库文件中的位置。PC 的文本文件中，汉字是用机内码的形式存储的，每个汉字占 2 字节，其中第一个字节为机内码的区码，汉字机内码的区码范围是从 0A1H（十六进制）开始，对应区位码中区码的第一区；而机内码的第二个字节为机内码的位码，范围也是从 0A1H（十六进制）开始，对应某区中的第一个位码。就是说将汉字机内码减去 0A0AH 就得到该汉字的区位码。例如汉字“北”的机内码是十六进制的“B1B1”，其中前两位“B1”表示机内码的区码，后两位“B1”表示机内码的位码。所以“北”的区位码为 0B1B1H－0A0A0H=1111H，将区码和位码分别转换为十进制，就可得到汉字“北”的区位码为“1717”。即“北”的点阵位于第 17 区的第 17



个字的位置，在文件 HZK16 中的位置为第 32×[(17−1)×94+(17−1)]=48640D 以后的 32 个字节为“北”的显示点阵。用 RF-1800 编程器读入二进制文件 hzk16j.bin 后利用其编辑功能中的缓冲区编辑查找到 BE00 H（48640D 是十进制，将其转变为十六进制后得到 BE00 H）开始的 32 个字节：04 80 04 80 04 88 04 98 04 A0 7C C0 04 80 04 80 04 80 04 80 04 80 04 80 1C 82 E4 82 44 7E 00 00（以上全为十六进制），将其写在 16×16 点阵方格纸上，即可得到如图 11-9 所示的点阵。

在应用时，连续取 32 个字节送到 LCD 的相应位置，就能正确显示汉字后的图形符号。在嵌入式系统中，如果需要显示的汉字是固定的，并且字数较小，可以直接在软件的头文件中定义汉字的显示点阵表格，要显示时，直接从表格变量数组中连续取出 32 个字节送往 LCD 显示器即可。但是如果要显示的汉字字数很多，或者又不能确定要显示哪些汉字时，使用上述办法就相当麻烦了，这时应当使用汉字字库。有些 LCD 驱动器自带汉字字库，如果没有带，可以自己使用 ROM 扩展，通过把汉字机内码送往字库文件，得到对应的显示点阵。

0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0
0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0
0	1	1	1	1	1	0	0	1	1	0	0	0	0	0	0
0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
0	0	0	1	1	1	0	0	1	0	0	0	0	0	1	0
1	1	1	0	0	1	0	0	1	0	0	0	0	0	1	0
0	1	0	0	0	1	0	0	0	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 11-9 “北”字的显示点阵

11.2.2 在嵌入式 Linux 中驱动 LCD

随着应用需求的推动，嵌入式 Linux 操作系统下也出现了许多图形界面软件包，如 MiniGUI、Trolletech 公司的 Embedded QT 等。至于 MiniGUI 软件包中的图形开发，在第 8 章中已对其进行专门的介绍。这些图形界面开发包与 WinCE 相似，在图形软件包的开发和移植工作中都牵涉到底层 LCD 驱动的问题。在本节中将介绍有关在嵌入式 Linux 下实现 LCD 驱动的相关知识。

前面已经阐述过 Linux 的设备管理是和文件系统紧密相关的，各种设备都以文件的形式存储在/dev 目录下，称为设备文件。应用程序可以打开、关闭、读写这些设备文件，完成对设备的操作，就像对普通数据文件操作一样。另外，Linux 把它所管理的设备分为字符设备和块设备，它们的区别在于系统为块设备提供了缓冲机制。由于涉及缓冲区的管理、

调度和同步等问题，块设备的驱动要比字符设备复杂。Linux把显示驱动也看作字符设备，把要显示的数据一个字节一个字节地送往LCD驱动器。

Linux为所有的设备文件都提供了统一的操作函数接口，方法是使用数据结构struct file_operations。这个数据结构中包括许多操作函数的指针，如open()、close()、read()、write()等。但是，由于外设的种类繁多，操作方式也不一样，如声音设备驱动要使用DMA通道，显示设备要提供对显存的操作，硬盘驱动要处理复杂的缓冲区结构等。如果file_operations中的函数都由驱动开发人员来编写实现，其工作量相当大，而且对于每一类设备，它的许多操作也是相似的，许多数据结构也是可以公用的。

为了解决这个问题，Linux采用了更高一层的封装方法，为同一类设备定义好了文件层次file_operations结构中的接口函数，在其中处理了大多数设备相关的操作，如各种缓冲区的申请和释放等。而具体操作底层硬件的一小部分则留给驱动开发人员去实现，这样就大大减少了驱动程序开发的难度。所以Linux提供了另外一个文件层到底层驱动的接口，通常为一个结构体，其中包含成员变量和函数指针等，不同的设备驱动有着不同的数据结构。这样，一方面保证了文件层I/O接口file_operations的一致性；另一方面，驱动程序的开发人员也不用了解设备驱动太多的细节，只需专注于硬件相关的I/O操作即可。

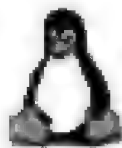
例如，一个具有代表性的声音设备，其文件层的file_operations定义如下：

```
struct file_operations oss_sound_fops={
owner:    THIS_MODULE,
llseek:   sound_lseek,
read:         sound_read,
write:    sound_write,
poll:         sound_poll,
ioctl:        sound_ioctl,
mmap:     sound_mmap,
open:     sound_open,
release:      sound_release,
}
```

其中，sound_read、sound_write等句柄指向的函数Linux都已提供，处理了许多与声音设备相关的操作，如DMA的申请、释放和操作等。而文件层到驱动程序的接口为audio_driver结构，其中包含底层操作函数，文件层的sound_read、sound_write会在需要时调用audio_driver中的函数。开发人员只需要编写audio_driver中的函数即可，这样就最大程度地减少了开发人员的工作量。

Linux为显示设备提供的帧缓冲驱动也采用这种"文件层—驱动层"的方式。下面首先对帧缓冲进行相应介绍。

FrameBuffer是出现在Linux2.2.xx及其以后版本内核当中的一种驱动程序接口，这种接口将显示设备抽象为帧缓冲区。用户可以将它看成是显示内存的一个映像，将其映射到进程地址空间之后，就可以直接进行读写操作，而写操作可以立即反映在屏幕上。



帧缓冲设备对应的设备文件是/dev/fb*，如果系统有多个显卡，Linux 还支持多个帧缓冲设备，最多可以有32个，分别为/dev/fb0、/dev/fb1、…、/dev/fb31。而/dev/fb 则指向当前的帧缓冲设备，一般为/dev/fb0。当然在嵌入式系统中一般只有一个帧缓冲设备。

帧缓冲设备也属于字符设备，与声音设备一样，也采用"文件层—驱动层"的接口方式。在文件层为之定义了数据结构：

```
static struct file_operations fb_fops={
        ower:           THIS_MODULE,
        read:           fb_read,    /*读操作*/
        write:          fb_write,   /*写操作*/
        ioctl:          fb_ioctl,   /*I/O 控制*/
        mmap:           fb_mmap,    /*映射操作*/
        open:           fb_open,    /*打开操作*/
        release:        fb_release, /*关闭操作*/
}
```

其成员函数都在 linux/driver/video/fbmem.c 中定义，其中的函数对具体的硬件进行操作，对寄存器进行设置，对显示缓冲进行映射。

由于显示设备的特殊性，在驱动层的接口中不但包括底层函数，还要有一些记录设备状态的数据，Linux 为帧缓冲设备定义的驱动层接口为 struct fb_info 结构，在 include/linux/fb.h 中定义，这个数据结构记录了帧缓冲设备的全部信息，包括设备的设置参数、状态以及操作函数指针等。每个帧缓冲设备都必须对应一个 fb_info 结构。其成员变量 modename 为设备名称，fontname 为显示字体，fbops 为指向底层操作函数的指针，这些函数需要驱动开发人员根据硬件设备的具体特性编写。另外还有两个记录设备状态的数据结构 struct fb_var_screeninfo 和 struct fb_fix_screeninfo，它们分别记录可设置信息和不可设置的设备信息。

如果要编写一个帧缓冲设备的驱动程序，开发人员所要做的主要工作包括：

- 编写初始化函数

初始化函数首先初始化 LCD 控制器，通过写寄存器设置显示模式和显示颜色数，然后分配 LCD 显示缓冲区。在 Linux 中可以用 kmalloc()函数分配一段连续的空间。缓冲区的大小为：点阵行数×点阵列数×用于表示一个像素的比特数/8。缓冲区通常分配在大容量的片外 SDRAM 中，起始地址保存在 LCD 控制寄存器中。

然后，初始化一个 fb_info 结构，填充其成员变量，并调用 register_framebuffer(&fb_info)，将 fb_info 登记入内核。

- 编写成员函数

编写 fb_info 中函数指针 fb_ops 对应的成员函数，对于嵌入式系统相对比较简单，只需实现以下3个函数即可：

```
struct   fb_ops{
```

```
......
int (*fb_get_fix)(struct fb_fix_screeninfo*fix,int con,struct fb_info *info);
int (*fb_get_var)(struct fb_var_screeninfo*var,int con,struct fb_info *info);
int (*fb_set_var)(struct fb_var_screeninfo*var,int con,struct fb_info *info);
......
};
```

struct fb_ops 在 include/linux/fb.h 中定义。这些函数都是用来设置/获取 fb_info 中的成员变量的。当应用程序调用 ioctl 操作时会调用它们。

通过/dev/fb，对显示设备的操作主要有以下几种：

- 读/写（read/write） /dev/fb

相当于读写屏幕缓冲区。例如 cp/dev/fb temp 相当于把屏幕中的内容复制到一个 temp 文件中，而 cp temp /dev/fb 则把 temp 文件中的内容在屏幕上显示出来。

- 映射（map）操作

由于 Linux 工作于保护模式，每个应用进程都有自己的虚拟地址空间，在应用程序中是不能直接访问物理缓冲地址的。为此，Linux 在文件操作接口 file_operations 中提供了 mmap 操作，可以将文件的内容映射到用户空间。对于帧缓冲设备，则可通过映射操作，将屏幕缓冲区的物理地址映射到用户空间的一段虚拟地址中（在 uClinux 中，没有 MMU，不存在虚拟地址，直接映射到实地址），之后用户就可以通过读写这段地址访问屏幕缓冲区，在屏幕上绘图了。

- I/O 控制

对于帧缓冲设备，通过对设备文件的 ioctl()调用可以读取/设置显示设备或者屏幕的参数，如分辨率、显示颜色数、屏幕大小等。Ioctl()的实现由底层驱动来完成。

在应用程序中，操作/dev/fb 的一般步骤如下：

（1）打开/dev/fb 设备文件。

（2）用 ioctl()取得当前显示屏幕的参数，如屏幕的分辨率、每个像素的比特数等，根据屏幕参数可以计算出屏幕缓冲区的大小。用于存储显示信息的有两个数据结构：struct fb_var_screeninfo 和 struct fb_fix_screeninfo。其中第一个存储可以设置的显示信息，如显示深度、分辨率等。第二个数据结构用于存储一些不可设置的显示信息，如屏幕缓冲区的物理地址、大小等信息。

（3）将屏幕缓冲区映射到用户空间。映射之后就可以直接对屏幕缓冲区进行读写、绘图和显示图片了。

典型应用程序如下：

```
#include<linux/fb.h>
main()
{
        int fbfd=0;
        struct fb_var_screeninfo vinfo;
```




```
        struct fb_fix_screeninfo finfo;
        long int screensize=0;
        unsigned char *fbp;
        /*打开设备文件*/
        fbfd=open("/dev/fb0",O_RDWR);
        /*取得屏幕相关参数*/
        ioctl(fbfd,FBIOGET_FSCREENINFO,&finfo);
        ioctl(fbfd,FBIOGET_VSCREENINFO,&vinfo);
        /*计算屏幕缓冲区的大小*/
        screensize=vinfo.xres*vinfo.yres*vinfo.bits_per_pixel/8;
        /*映射屏幕缓冲区到用户地址空间*/
        fbp=(unsigned char *)mmap(0,screensize,PORT_READ|PORT_WRITE,
MAP_SHARED,FBFD,0);
        /*下面可以通过 fbp 访问缓冲区，进行图形绘制*/
        ……
}
```

11.3 在嵌入式 Linux 中使用 LCD

在前面已经介绍了 LCD 的原理和分类，以及在嵌入式 Linux 下如何驱动点阵式 LCD。本节中将以在华恒 HHDREZ328_R2 开发板上实现扩展 LCD 显示器的应用为例，介绍如何在嵌入式 Linux 下使用 LCD 显示器。

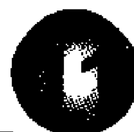
11.3.1 EZ328 对 LCD 的支持

在此使用华恒 HHDREZ328_R2 开发板，华恒 HHDREZ328_R2 开发板所使用的微处理器为 Motorola 公司的 MC68EZ328。它为 LCD 的控制提供了很好的支持，其实现主要是通过 LCD 控制器完成的。LCD 控制器负责为外部的 LCD 驱动器提供数据，它支持的最大分辨率在黑白方式下为 640×512，在灰度方式下为 320×240，支持 16 级灰度。它的总线宽度非常窄，支持 4bit、2bit 或者 1bit LCD 总线带宽。

通过周期性的 DMA 传送，LCD 控制器从系统内存中读取显示数据送往 LCD 驱动器。由于使用很窄的总线带宽，所以可以给予内核充分的处理时间。LCD 控制器由中断寄存器组、逻辑控制模块、行 FIFO 缓冲以及光标逻辑块等构成，如图 11-10 所示。

MPU 接口寄存器组启动 LCD 控制器的各种控制功能。它们与 68K 的总线相连，利用控制逻辑模块为其他模块提供内部控制和计算信号。DMA 向核心发出总线请求，当得到总线控制权后，它将执行一些内存突发操作，把内存中的显示数据搬到行 FIFO 缓冲中。每一个脉冲所需的 DMA 时钟周期数目是可以修改的，这样很容易使系统具有不同的存储速度。

行缓冲在 DMA 时钟周期内将数据从内存中读出来，然后将其送入光标逻辑模块。输



入与较快的 DMA 时钟同步，而输出与较慢的 LCD 时钟同步。光标控制逻辑用来设置显示屏幕上的块状光标，可以调节光标的高度与宽度，光标的亮度和刷新率都可以通过设置 LCD 闪烁控制寄存器来调节。

帧速率主要用来控制灰度等级，并能为最大为 16 个灰度级别设置 16 个灰度标度。灰度等级对应于当画面刷新时一个像素的打开次数。由于液晶的显示亮度随驱动电压而变化，通过对 LCD 灰度调节控制器（LGPMR）的编程可以很好地调节灰度。

LCD 接口逻辑将要显示的数据打包成合适大小，并传送到 LCD 控制面板的数据总线。通过编程可以很好地控制诸如 LFLM、LP、LCLK 等信号，以迎合不同的 LCD 的需求。

至于 LCD 控制器与外部 LCD 驱动器相连的信号引脚，以及 LCD 控制器中的各个寄存器的含义，如何对它们进行编程设置等内容，请参看 MC68EZ328 用户手册。

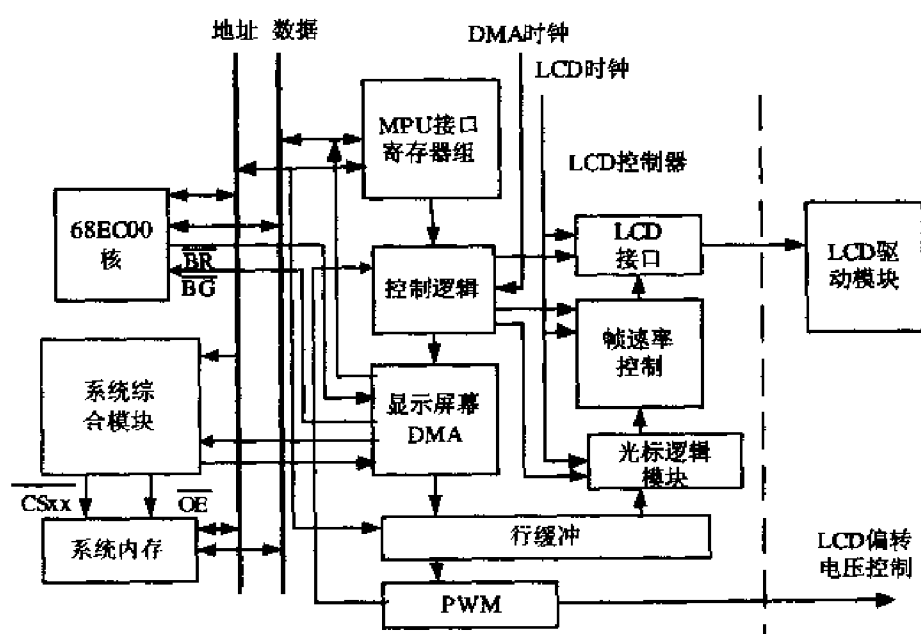


图 11-10　LCD 控制器结构图

注意：MC68EZ328 对 LCD 控制器的各个寄存器分配了固定的端口地址，如 LCD 起始寄存器被分配的端口地址为 0xFFFFFA00，LCD 虚拟页面宽度寄存器的端口地址为 0xFFFFFA05 等。

11.3.2　uClinux 对 LCD 显示器的支持

在软件方面，uClinux 已经提供了对 LCD 显示器的支持。它表现在启动时初始化 LCD 显示器和提供绘图 API 两个方面。

1. 启动初始化

在华恒套件附带的 linux-2.0.x 内核中，编译开关（INIT_LCD）默认是关闭的，也就是

在启动初始化代码文件/linux/arch/m68knommu/platform/68EZ328/ucsimm-head.S 中，使用了#undef INIT_LCD 语句。如果要使用 LCD 显示器，应当在该文件中把这个开关打开，再重新编译后即可。而在 linux-2.4.x 内核中，默认的 LCD 显示器的编译开关是打开的，在初始化文件/linux/arch/m68knommu/platform/68EZ328/ucsimm/crt0_fixed.S 中定义了如下语句：#define CONFIG_INIT_LCD。下面以 2.4 内核为例介绍对 LCD 显示器的初始化过程。

首先，对寄存器进行设置。其代码为：

```
#if 1
/*在 LCD.h 中已经定义 CONFIG_INIT_LCD 为 1，这个条件编译就是判断开关状态，如
果编译开关已经打开，即可完成 LCD 的初始化*/
    movel    #splash_bits, 0xfffffA00     /* LSSA，即 LCD 显示屏幕起始地址寄存器*/
    moveb    #0x28,     0xfffffA05        /* LVPW，虚拟页宽寄存器*/
    movew    #0x280,    0xFFFFFa08        /* LXMAX，LCD 宽度寄存器*/
    movew    #0x1df,    0xFFFFFa0a        /*LYMAX，LCD 高度寄存器*/
    moveb    #0,        0xfffffa29        /*LBAR，LCD 刷新速率调整寄存器*/
    moveb    #0,        0xfffffa25        /* LPXCD，LCD 像素时钟除数寄存器*/
    moveb    #0x08,     0xFFFFFa20        /* LPICF，LCD 板接口设置寄存器*/
    moveb    #0x01,     0xFFFFFA21        /*设置送往 LCD 面板的信号的极性*/
    moveb    #0x81,     0xfffffA27        /* LCKCON，这些位表示静态显示内存的等待
状态控制，它的数值为每个 DMA 访问周期的时钟数目，这里定义一个时钟周期*/
    movew    #0xff00, 0xfffff412          /* LCD 引脚，0xfffff412 对应的是 PC 口*/
#endif
```

另外在文件中，还应包含如下语句：

```
#ifdef CONFIG_INIT_LCD
splash_bits:
#include "bootlogo.rh"
#endif
```

其中，bootlogo.rh 文件是启动时显示画面的资源文件，在此要把它包含进来。该文件表示的画面是一个 Linux 的代表宠物：企鹅。

然后，进行如下操作进行配置：

#cd /linux

#make menuconfig

General setup--------→

[*]Console support 打开

[*]Frame buffer 打开

这样在启动开发板后，LCD 显示屏将显示一个 Logo——一只小企鹅。它实现的方法就

是直接写LCD的显存。初始化过后，系统在/dev下将建一个设备文件fb0。可以对该设备进行读写，等同于对LCD显示屏的操作。

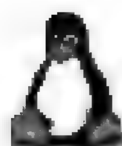
2. 图形API的实现

在华恒的EZ328开发套件中，提供了一系列的图形界面接口函数，它们利用底层的LCD驱动程序屏蔽了底层的硬件细节，使用户能够把注意力集中到图形应用程序的编写上，很快地熟悉这些图形API并使用它们开发图形界面。这些函数都是在/user/gui/graphic.c中实现的，下面将结合前面的驱动程序编写，介绍在华恒的开发套件中是如何实现这些API的。

首先是初始化图形界面initgraph()，它应该在使用别的图形API前调用。其作用主要是打开帧缓冲设备文件，获取显示设备的信息，进行显示缓冲的内存映射，然后打开字库文件，获取文件指针，以显示字符，它的完整代码如下所示：

```
short initgraph(void)
    {
      /*定义用于获取显示设备信息的数据结构*/
  struct fb_var_screeninfo screeninfo;
      /*打开帧缓冲设备文件*/
    screen_fd = open("/dev/fb0", O_RDWR);
  if (screen_fd == -1)
      {
          perror("Unable to open frame buffer device /dev/fb0");
      return 0;
       }

      /*获取设备信息*/
    if (ioctl(screen_fd, FBIOGET_VSCREENINFO, &screeninfo)==-1) {
      perror("Unable to retrieve framebuffer information");
      return 0;
          }
      /*把屏幕的宽度和高度赋给全局变量*/
screen_width = screeninfo.xres_virtual;
       screen_height = screeninfo.yres_virtual;
       /*ASC码字库函数指针*/
E_Font    = (unsigned char*)(screeninfo.english_font);
       //printf("E_Font Address %x %x\n",E_Font,screeninfo.english_font);
       if(!E_Font)
        //E_Font=(unsigned char*)(0x8804);
        //E_Font=(unsigned char*)(0x8812);
        E_Font=(unsigned char*)(0x0004a690);
       /*进行显示缓存映射。缓存的大小为screen_height * screen_width>>3，也就是高度
×宽度÷8，screen_ptr指向显存的起始位置*/
```




```
screen_ptr = mmap(0, screen_height * screen_width>>3,
           PROT_READ|PROT_WRITE, 0, screen_fd, 0);

if (screen_ptr==-1) {
              perror("Unable to mmap frame buffer");
           close(screen_fd);
            return 0;
         }
      /*打开汉字字库文件，CHINESE_FONT_FILE 在文件的开头已经宏定义为字库文
件/font/hanzi */
  C_Font=fopen(CHINESE_FONT_FILE,"rb");

  if(!C_Font)
      {
        perror("Unable to open Chinese font file");
        close(screen_fd);
        return 0;
      }
  Color=1;
  return 1;
}
```

在初始化图形界面后，就可以显示字符或者图形了。下面首先介绍显示字符的 API，这个函数名为 textout()，它的功能是在指定的位置显示一串字符串，其函数实现如下所示：

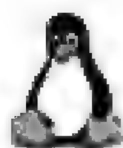
```
void textout(short x,short y,unsigned char *buf)
{
  int i,j,count=strlen(buf);
  short k,l,m=screen_width>>3;
  char pixel[32];
  //printf("screen_height=%d\n",screen_height);
  #ifdef FONT
  if(!font) {
            puts("Please Init Chinese Environment First!");
                  return;
              }
#endif

   for(i=0;i<count;)
   {
    if(y>=screen_height) /*指定的 y 坐标大于屏幕高度，需要刷屏*/
         {
```

```
memmove(screen_ptr,screen_ptr+320,2880); /*把下一行的内容往上移一行*/
   memset(screen_ptr+2880,0,320);/*原来的内容置空*/
y-=16;
  }

if((buf[i]>=161)&&(buf[i+1]>=161)) /*要显示的是汉字，buf[i]和 buf[i+1]表示一个汉字*/
  {
   j=((buf[i]-161)*94+(buf[i+1]-161))<<5;/*得到汉字的区位码*/
   fseek(C_Font,j,SEEK_SET);
   fread(pixel,32,1,C_Font); /*得到汉字的点阵数组*/
       draw_bmp(x,y,2,16,pixel); /*显示汉字*/
   x+=16;
/*指定的 x 坐标大于屏幕宽度，需要换行*/
 if(x>=screen_width) {
                    y+=16;
                    x=0;
                   }
 i+=2; /*汉字占用两个字节*/
  }
else/*显示 ASC 字符*/
 {
  if(buf[i]=='\n') {/*要显示的是换行*/
           k=(y*screen_width+x)>>3;
       l=(screen_width-x)>>3;
           for(j=0;j<16;j++,k+=m)
             memset(screen_ptr+k,0,l);
           x=0;y+=16;
           }
 else
 {
      draw_bmp(x,y,1,16,E_Font+(buf[i]<<4)); /*把 ASC 字符的点阵数组送往显示
函数显示*/
      x+=8;
     }
    i++;/*ASC 字符占用一个字节*/
   }
  }
 }
```

在以上函数实现中调用了显示字符点阵的函数 drawbmp()，这个函数显示的是以二进制形式表示的 pixel 数组中的点阵。它的调用形式为 void draw_bmp(short sx, short sx, short rwidth, short height, char* pixel)，其中 sx、sx 为显示的起始位置，rwidth、height 为要显示



的宽度和高度，pixel 中的数据为二进制表示的点阵。该函数的实现代码如下所示：

```
    void draw_bmp(short sx, short sy, short rwidth, short height, char* pixel)
{
     short i, j, k, t,l=rwidth*height,m;
     int off,d;
     /*在显示缓冲区中的起始位置*/
     char *loc=screen_ptr+(off=((sy*screen_width+sx)>>3));
     //显示
          for(t=0,k=0,i=0;i<height;i++,k+=screen_width>>3,t+=rwidth)
           for(j=0;(j<rwidth)&&(j<20);j++)
     {
              m=k+j;
                 if(m+off>=3200)
     //  if(m+off>=4000)
                       return;
     /*把要显示的点阵数据送往显示缓冲的对应位置，LCD 控制器会通过 DMA 传送
把数据送往 LCD 的相应位置显示*/
              loc[m]=pixel[t+j];


     }
}
```

以上介绍的是如何显示字符，其实显示图形所遵照的原理与它是一样的，最终都是把要显示的图形的点阵数据送到显存的对应位置。由于篇幅原因，在这里就不再对每一个函数都作介绍了。

11.3.3 图形 API 使用实例

对于前面讲述的图形 API，本节将通过实例介绍其具体使用方法。本实例是华恒 HHDREZ328_R2 开发套件中/user/gui 下面的一个例程，它将完成清屏、移动翻滚地显示"华恒科技" 4 个字等功能。

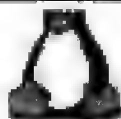
首先，在这个头文件中，引用了图形 API 函数和一些数据结构（见光盘/demo/chap11/11-2）：

```
/*
 * $Id: gui.h, v 1.0 2001/02/12 19:03:26 till Exp $
 *
 * basic gui header file
```

```
 *
 * Copyright (C) 2001    Chen Yang <chyang@hhcn.org>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 */
#ifndef GUI_H
#define GUI_H

typedef unsigned int UINT;
/*定义 bmp 图形的头文件*/
typedef struct{
        unsigned char id[2];
        long filesize;
        short reserved[2];
        long headsize;
        long infosize;
        long width;
        long height;
        short planes;
        short bits;
        long biCompression;
        long sizeimage;
        long biXpp;
        long biYpp;
        long biclrused;
        long biclrimportant;
}BMPHEAD;

/*该联合定义了显示模式。在 setmode()中用来设置显示模式，它的含义为：MODE 表示模式，SRC 表示源，NOT 表示取反，XOR 表示或，DST 表示目的，如 MODE_SRC_OR_NOT_DST 就表示目的取反再与源相或后的结果*/
typedef enum {
  MODE_SRC,
  MODE_NOT_SRC,
  MODE_SRC_OR_DST,
  MODE_SRC_AND_DST,
  MODE_SRC_XOR_DST,
  MODE_SRC_OR_NOT_DST,
```




```
  MODE_SRC_AND_NOT_DST,
  MODE_SRC_XOR_NOT_DST,
  MODE_NOT_SRC_OR_DST,
  MODE_NOT_SRC_AND_DST,
  MODE_NOT_SRC_XOR_DST,
  MODE_NOT_SRC_OR_NOT_DST,
  MODE_NOT_SRC_AND_NOT_DST,
  MODE_NOT_SRC_XOR_NOT_DST,
  InvalidMode
} CopyMode;

/*该联合定义填充的模式*/
typedef enum {
   BlackPattern = 0,
   WhitePattern,
   DarkGreyPattern,
   LightGreyPattern,
   MicroPengPattern,
   InvalidPattern
} PatternIndex;
/*下面为引用的各个 API 函数，具体实现请参考 Minigui 的源代码*/
extern unsigned char *screen_ptr;
extern short initgraph(void);
extern void closegraph(void);

extern void clearscreen(void);

extern void setpixel(short x,short y,short color);
extern short getpixel(short x,short y);

extern void setmode(CopyMode mode);
extern CopyMode getmode(void);

extern void setcolor(short color);
extern UINT getcolor(void);

extern void setfillpattern(PatternIndex index);
extern PatternIndex getfillpattern(void);

extern void bar(short x1,short y1,short x2,short y2);
extern void ellipse(short x1,short y1,short x2,short y2);
extern void line(short x1, short y1, short x2,short y2);
```

```
extern void lineto(short x1, short y1);
extern void moveto(short x,short y);
extern void rectangle(short x1,short y1,short x2,short y2);
extern void textout(short x,short y,unsigned char *s);

extern void V_scroll_screen(short height); //Vertical Scroll ^
extern void H_scroll_screen(short width);    //Horizontal Scroll:<-->

#endif
```

以下是函数的主体实现代码：

```
/* gui.c: Graphics demos
 *
 *      Programmed By Chen Yang (support@hhcn.com)
 *
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 */
#include "gui.h"
main(int argc,char*argv[])
{
    short i,j,w,h;
    PatternIndex p=BlackPattern;/*填充模式为黑色模式*/
    char buf[512];
    if(initgraph())/*初始化图形界面*/
    {
    if(argc>1)
    {
        ShowBMP(argv[1]);
        for(i=1;i<160;i++)
        {
              V_scroll_screen(1);/*以 1 为单位向上滚动屏幕*/
        }
/*本程序的使用方法为：gui xx.bmp，而且这个位图文件应该是黑白两色的，否则，将显示：Unsupported color bitmap!的出错信息*/
        ShowBMP(argv[1]);
        for(i=1;i<160;i++)
        {
```




```
            V_scroll_screen(-1);/*以 1 为单位向下转屏*/
    }
      textout(0,0,"Press Enter To Show File.");
      ShowBMP(argv[1]);

      for(i=1;i<160;i++)
      {
        H_scroll_screen(1);/*以 1 为单位水平向左转屏*/
      }
      ShowBMP(argv[1]);
      for(i=1;i<160;i++)
      {
        H_scroll_screen(-1); /*以 1 为单位水平向右转屏*/
      }
    }
    clearscreen();/*清屏*/
    for(;p<InvalidPattern;p++)
    {
    setfillpattern(p);/*设置各种填充模式*/
    sprintf(buf,"%d",p); /*将填充模式输出到数组 buf 中*/
    textout(120,0,buf); /*在（120,0）处打印输出模式*/
    fillrect(0,0,120,120); /*以该模式填充（0,0,120,120）的矩阵*/
      }
     clearscreen();

    textout(0,0,"华恒科技");

    for(i=0;i<16;i++)
        /*复制显存，"华恒科技" 4 个字要用 8 个字节来表示，显存起始地址由 LCD 起
始地址寄存器 LSAA 定义，此处为 0x400*/
        memcpy(buf+i*8,0x400+i*20,8);

  for(i=0;i<16;i++)
      memcpy(0x400+320+i*20,buf+i*8,8);

  srand(0);/*产生一个随机数*/
   for(;;)
   {
      i=rand()%160;
      j=rand()%160;
      if(i<96)
        if(j<144)
```

```
/*把要显示的数据按规定的方式输出到LCD屏幕上，其调用方式为*/
/*void bitblt(
        short src_x,
        short src_y,        ///////// src_x,src_y 为像素在LCD上的源地址

        short w,
        short h,        //////数据块的宽度和高度

        short dest_x,
        short dest_y,  //////数据块将要传送到的目的地址
        unsigned char *src,////////指向源数据块的指针
        short src_units_per_line, //////数据源每行的宽度
        unsigned char *dest,///////指向目的地址的指针
        short dest_units_per_line//////目的数据块每行的宽度
        )*/
       bitblt(0,0,64,16,i,j,buf,8,0x400,20);
  else
     if(j>=144)
   bitblt(0,0,64,159-j,i,j,buf,8,0x400,20);
  else
     if(j<144)
        bitblt(0,0,159-i,j,i,j,buf,8,0x400,20);
  else
     if(j>=144)
   bitblt(0,0,159-i,159-j,i,j,buf,8,0x400,20);
 }
 closegraph();/*显示完毕*/
 }
}
```

本程序的输出结果为：当输入 gui 命令时，在 LCD 上将会出现"华恒科技"4 个字，而且其出现的位置是随机的。当输入的是 gui xxx.bmp 命令，那么首先会在 LCD 上显示输入的位图，然后清屏，再在屏幕上显示"华恒科技"4 个字。

11.4 小结

本章介绍了嵌入式系统中人机界面的输出问题。在嵌入式系统中，用作输出的可以是段式的 LED，也可以是功能更为强大的 LCD，当然 LCD 也可以是段式的。对于 LED 显示来说，有静态显示和动态扫描显示之分，它们的特性以及各自与微处理器的连接方式，在



本章已经作了详细的介绍。特别是对于动态扫描显示 LED 的方式，在书中给出了实例代码说明。

对于功能复杂的嵌入式应用系统，更多地需要扩展 LCD 显示器。本章重点对这方面作了介绍，包括 LCD 的显示原理、LCD 的分类、LCD 显示器的驱动等内容。其中，在嵌入式 Linux 中驱动 LCD 是开发人员所要做的主要工作，本章花了大量的篇幅对这方面进行了介绍。最后，还以华恒开发板中的 LCD 模块为例，介绍了如何在嵌入式 Linux 下扩展 LCD 显示器，并进行图形界面开发等相关知识。

11.5 思考题

1．试说明 LED 的发光原理。

2．LED 的显示方式有哪些？各有什么优缺点？

3．试说明使用 LED 译码器与不使用 LED 译码器之间扩展 LED 显示器的区别。

4．试说明 LCD 的显示原理和 LCD 的分类。

5．在 LCD 显示器中是如何显示汉字的？

6．什么是 buffer frame?帧缓冲设备是字符设备吗？

7．请说明驱动帧缓冲设备的过程，它与前面讲的驱动标准字符设备有什么区别？

8．在华恒开发板中，当上层的图形 API 应用到下层的 LCD 驱动程序时，它是如何调用下层的驱动程序的？

第 12 章　在嵌入式 Linux 系统中扩展 PCI 设备

知识点：

- PCI 总线的特征
- PCI 配置空间
- 嵌入式 Linux 中对 PCI 设备的支持
- 嵌入式 Linux 下的 PCI 驱动程序的编写

本章导读：

本章将主要介绍如何在嵌入式 Linux 系统中扩展使用 PCI 总线设备。作为 PC 机上目前占统治地位的标准总线，PCI 总线在嵌入式系统中也得到广泛的应用。它是一种将系统中外部设备以结构化与可控制方式连接到一起的总线标准，主要用于大数据量的高速通信场合。

本章将首先对 PCI 规范加以介绍，包括 PCI 总线的特征、使用 PCI 总线的系统结构，PCI 的配置空间等，以使读者对 PCI 总线规范有一个初步的认识。然后，将介绍在嵌入式 Linux 中，是如何支持 PCI 总线的，包括如何发现系统中的 PCI 设备和如何为每个设备分配资源等。最后，将介绍在嵌入式 Linux 下如何实现 PCI 设备的驱动，主要包括如何查找到要驱动的设备、如何获取分配给 PCI 设备的资源信息、如何注册设备驱动程序等。通过本章的学习，读者将会掌握在嵌入式 Linux 下使用 PCI 设备的相关知识。



12.1 PCI 总线规范

传统的计算机总线（如 ISA、EISA 和 MCA 等）由于带宽的限制，已经成为制约计算机性能提高的瓶颈，不能满足外部设备与 CPU 高速通信的需要。ISA 只有 8 位和 16 位两种通信方式，最高传输速率只有 8Mbit/s；EISA 是扩展的 ISA 总线，虽然能支持 32 位数据，32 位地址，但其成本较高，主要用于计算机的服务器领域，不适合嵌入式领域；而 IBM 的微通道（MCA）可以认为是一种标准总线，但由于其专利的封闭性而难以广泛流行。

PCI（Peripheral Component Interconnection，外部设备互连）总线的出现解决了这一问题，PCI 总线是 32 位的并且可以升级到 64 位，它独立于 CPU，总线时钟高达 33/66MHz，使用同步控制、突发传送（burst）等手段可以使总线数据传输速率高达 132Mbit/s（32 位总线），或者 264Mbit/s（64 位总线）。适用于外部设备与 CPU 的高速连接，如视频信号采集卡中把外部视频信号经过编码再用 PCI 总线送入 CPU 处理，或者用于交换机的交换模块与 CPU 的连接等。接下来，首先对 PCI 总线规范进行相应的介绍。

12.1.1 PCI 总线规范简介

1. 总线速率

PCI 总线首先是为了解决 PC 机中外部设备与处理器之间数据传输的速度瓶颈而制定的，早期的 ISA 乃至 EISA 的一大缺点就是速度太慢，不能满足图像、网络等方面的需要。为此，PCI 规范从多个方面采取了相应措施以提高系统总线传输速率，主要有：

- 使用 32 位乃至支持扩展的 64 位总线带宽。
- 驱动总线的时钟频率提高为 33MHz，对于扩展的 PCI 总线，还支持 66MHz 的时钟频率。
- 使用突发传输方式。采取突发传输就是只寻址一次，但可以进行对同一寻址设备多次数据发送。一次突发传输由一个地址段和相随的一个或多个数据段组成，最大的突发数据段是 256 个。

采用以上措施，可使 PCI 的总线速率达到 132Mbit/s，而对于增强的 64 位总线结构的 PCI，传输速率可以高达 264 Mbit/s。

2. 主设备和从设备

PCI 设备有主从之分，主设备占有 PCI 总线，是数据传输的发起者，它需要首先申请 PCI 总线，在得到批准后，再往 PCI 地址线上送出要寻址的从设备的地址，等从设备准备好后，就可以发送或者读取数据了。从设备也叫目标设备，在数据传输中它是被动的，它按照寻址要求，如果准备好则使能特定信号，然后等待主设备的读或者写要求。

在 PCI 传输过程中，主设备控制目标设备的传送，要发起数据传输，主设备应该使能



必要的总线信号并保持有效直到它们被目标设备认可。要表明传送完成，目标设备应该提供终止周期，其中可能包括错误或者重试信号。PCI 接口自动插入必要的等待以防止主设备时间溢出，保证高速的 PCI 传送能在速度相对较低的目标设备上进行。

作为数据传输的发起者，一个 PCI 总线主设备必须能够执行突发读和写，对存储器、I/O 和配置空间进行寻址，响应系统复位要求，能够进行总线上的奇偶校验，报告奇偶校验错误，还必须能够识别目标设备的终止、重试信号和在时间溢出时能够停止数据传输。而目标设备应该能够进行地址解码、处理配置操作、响应系统复位、产生奇偶校验、报告奇偶校验错误和系统错误、产生重试和目标设备终止信号等。

3. 总线仲裁

由于在一个计算机系统中可能有多个 CPU 主设备，它们有可能在同一时间都需要占用 PCI 总线，这将发生 PCI 总线的竞争问题。在早期的计算机系统中，系统中只有 CPU 才是主设备，而其他设备都是从设备，只有 CPU 才能启动跨总线的操作。当然也有例外，就是外部设备对内存的 DMA 操作。这是外部设备先向 CPU 发出 DMA 请求，让 CPU 暂停访问内存，实际上是暂停包括访问总线在内的一切外部操作，使系统中暂时没有了“主设备”，得到允许后，外部设备（“从设备”）就暂时升级变为“主设备”了，从而可以直接对内存进行操作，这种操作当然也是跨总线的。但是由于 CPU 已暂停活动，所以不存在总线竞争的问题。随着计算机技术的发展，一些“从设备”，即外部接口卡也带上了智能，或者说带上了本地的处理器，在这种情况下，应该允许一个外部设备直接访问另一个外部设备的存储器和 I/O，而不必 CPU 介入。

但是系统中的 PCI 总线只有一条，外部主设备要实现对从设备的访问，那当然要先取得对 PCI 总线的使用权。为了解决竞争使用总线的问题，在 PCI 总线上配备了总线仲裁器。外部主设备如果要使用 PCI 总线，首先发出总线申请信号，如果这时没有其他设备发出请求信号，那它就自然地占有了系统总线。如果遇到冲突，仲裁器会选择其中之一（包括 CPU）暂时成为主设备，并向它发出肯定的答复，而其他设备只有等待。但是对于有些写的操作，如果是 CPU 往外部写数据，由于总线冲突一时不能成为主设备，为了不让它在原地等待而影响效率，PCI 采取的措施是让它把要写的内容放到一个缓冲区中，交付给 PCI 总线，待申请到总线后再从缓冲区中往外写数据，而 CPU 则可以继续执行，无需在原地等待。当然对于读操作，就无能为力了。所以，从效果来看，跨 PCI 总线的写操作往往比读操作快。

为了减小操作延迟，PCI 仲裁是基于操作而不是基于时间片的。也就是如果 PCI 主设备获得一次总线控制权，它可以完成一次时间不确定的操作，而不是一个规定的时间片。PCI 使用了一个中央仲裁电路，系统中每个主设备都有一个惟一的请求（REQ#）和允许（GNT#）信号与之相连。仲裁是“隐蔽”的，也就是说它发生在上一个总线操作期间，所以不会因为等待仲裁而耗费 PCI 总线周期。

总线仲裁采用一种特定的仲裁算法作为仲裁的基础，例如循环算法、优先级算法、公正算法等。由于采用哪种算法在 PCI 规范中并没有确定，所以系统的设计者可以根据系统的实际情况而确定。但是一个特定的系统的算法是确定的，它们是产生错误时延的基础，

所以必须提供它们所选用的 I/O 控制器和外插卡的延迟要求。总线允许对优先的和重要的请求采取灵活的响应方式。在任何周期，只要 GNT#信号有效，仲裁器就提供给某一个外部设备总线控制权。

外部设备通过时期 REQ#信号有效来请求总线，但是它决不能用 REQ#将自己“停放（park）”在总线上，如果使用了总线停放，那么它应该已经是仲裁器指定的 PCI 总线拥有者。当仲裁器判断某个外部设备可以使用总线时，它使该外部设备的 GNT#有效。如果当前总线拥有者要求额外传送，应该保持 REQ#有效，以申请连续传送。这时如果没有别的设备发出总线请求信号，或者当前总线主设备具有最高的优先级，仲裁器将让其继续操作总线。

4. 奇偶校验和错误处理

PCI 上的奇偶校验提供了一种机制，以检查总线主设备是否成功地寻址所希望的目标设备，或者它们之间的数据传送是否正确。在地址段和数据段期间，无论所有 AD[31..0] 及 C/BE[3..0]线是否都带有有意义的信息，它们都要参与奇偶校验。未传送数据的字节通道也要求驱动到稳定状态并被包括在奇偶校验中。即使在配置周期、特殊周期或者中断应答命令中，一些（或者全部）地址线未定义，也都要求驱动到稳定状态并且接受奇偶校验。

所有总线主设备和目标设备都要对从 PCI 总线上获得的地址和数据作奇偶校验并报告奇偶校验错误。为进行奇偶校验，设备要实现以下功能：

- 设备内部通过锁存 AD[31..0]和 C/BE[3..0]一起进行异或操作产生奇偶校验。
- 在下一个时钟脉冲，设备将它的内部计算结果和产生偶校验的 PAR 进行异或操作。
- 如果这两个值一致，偶校验没错。否则，在下一时钟脉冲要么通过使能 PERR#报告数据有错，要么通过使能 SERR #报告地址错误。

当 PCI 检测到奇偶校验或者其他系统错误时，从 PCI 设备硬件操作到设备驱动程序、设备管理器，再到操作系统都要对之进行相应处理，以对错误进行弥补，如对数据进行重新访问或者向上层报告错误。

在 PCI 错误反应设计中，使用了两个信号——PERR#和 SERR#（引脚），PERR#专门用于报告除特殊周期命令外的所有传送中的数据奇偶校验，它是一个持续三态信号，总线协议保证 PERR#不会同时被多个总线设备驱动，并且适当的信号转换时间也避免了多个驱动器争用。只有总线主设备才能反映读数据奇偶错误，而只有选中的目标设备才能反映写数据的奇偶错误。

不管在什么情况下，当支持奇偶校验的外部设备在检测到奇偶校验错误时，它都要设置配置空间中的状态寄存器的检测到数据奇偶错误位（Parity Error Detected）。对奇偶校验位错误的信号发生和应答都是由命令寄存器中的奇偶错误应答位（Parity Error Response）来控制，如果此位被清除，该设备就忽略所有的奇偶校验错误，并认为奇偶正确而传送完成。如果设置了该位，则外部设备检测到奇偶错误时会使 PERR#信号有效。PCI 配置空间的详细介绍请参看下一节。

当一个总线主设备检测到数据奇偶错误并使 PERR#有效（在读操作中）或者采样到 PERR#有效（在写操作中）时，它必须设置检测到数据奇偶错误位（状态寄存器的第 8 位），这时它可以继续传送也可以终止传送，需要注意的是只有主设备才能设置该位，目标设备



可以选择继续完成本次操作或者发送终止信号，但是不能设置状态寄存器中的检测到数据奇偶错误位。对目标设备而言，PERR＃只是一个输出信号，而对总线主设备来说，它可以是输入和输出信号。

如果一个正在执行操作的主设备发现在它的传送中发生了奇偶校验错误，它应该报告给系统。建议采用中断的方法让总线主设备通知其设备驱动程序（或调整状态寄存器，或标志）。如果这些方法无效，可以使 SERR＃有效，以便可靠地将错误信息传送给操作系统处理。

注意：有些系统设计人员会将所有的 PERR＃错误转换成 SERR＃错误，以便将奇偶错误信息传送给操作系统。

SERR#用于发出所有地址奇偶校验错误和特殊周期命令中的数据校验错误，并且可能有选择地用于别的非奇偶性的系统错误中。它是漏极开路输出，并且所有 PCI 设备的 SERR＃线可以连到一起，所以它可能同时被多个 PCI 设备驱动。因为漏极开路信号在一个时钟内不能稳定，所以至少要连续两个时钟沿采样到 SERR＃有效，才能判断 SERR＃出错。无论是主设备还是目标设备都可以设置 SERR＃有效。但是，要使 SERR＃有效首先得使配置空间的命令寄存器的 SERR＃允许位置为 1。

检测到地址奇偶错误的 PCI 设备将作下列之一的处理：

- 确认总线周期并像地址正常一样终止传送。
- 确认总线周期并用目标设备失败，终止传送。
- 不确认总线周期，任由传送主设备以失败而终止。这时目标设备不允许重试或解除连接终止传送，因为已经检测到地址奇偶错误。

所有外部设备都要求进行奇偶校验，并且要求报告奇偶校验错误，而对于非奇偶校验的 SERR＃信号是可选的。而且必须注意到 SERR＃上发出信号将产生 NMI，所以，在使用 SERR＃时要小心行事。

5. 系统结构

采用 PCI 总线后，现代微机普遍采用南北桥结构。其中北桥是一个 HOST-PCI 桥，它靠近 CPU，桥连接微处理器和 PCI 总线。它是一个 PCI 总线控制器，包括 PCI 总线仲裁器等。另外，微处理器一般还要通过内存控制器经常访问内存，所以通常把内存控制器也做到北桥芯片中。CPU 与北桥的连接，以及通过北桥芯片的内存控制器与内存的连接就是传统意义上的“系统总线”。所有的外设，包括磁盘、键盘、显示器、显卡、串口、网卡等，全部都直接或间接地连在 PCI 总线上，CPU 对外部设备的访问都要通过北桥。

另外，在计算机中还有许多设备并不是标准的 PCI 设备，可能不支持 PCI 接口。南桥芯片则负责提供系统外部设备（如 IDE、USB 以及 Super I/O 等）接口的支持，把它们连接到 PCI 总线上。南桥远离微处理器，主要执行设备接口转换工作。

还有一个问题就是 PCI 总线的扩展问题，直接与北桥芯片相连的 PCI 总线为主（Primary）PCI 总线，它的总线号为 0。由于在主总线上能够连接的 PCI 设备有限，将不能满足外部 PCI 设备扩展的需要。在 PCI 规范中，使用 PCI－PCI 桥来实现 PCI 总线扩展。利用一个 PCI—PCI 桥，可以在一条 PCI 总线下再扩展一条 PCI 总线，例如在 PCI 总线 0



下扩展一条 PCI 总线，它的总线号为 1。利用 PCI—PCI 桥，在不同 PCI 总线上的设备之间的访问都是透明的，也就是它们感觉到好像在同一总线上一样，只是寻址的地址信号存在一定的差别。而且在下层 PCI 总线上还可以扩展 PCI 总线，一层一层形成一个树型结构。对于上层总线来讲，连接在总线上的 PCI 桥也是它的一个设备。但它是一种特殊的设备，它既是上层总线的设备，又是它的延伸。

在 PC 机中有许多外设，既有高速设备如显卡、网卡等，也有低速设备如使用 ISA 总线的声卡、串口等，在 PC 机的体系结构中为了实现对使用老式 ISA 总线的设备进行兼容，使用了 PCI—ISA 桥。PCI—ISA 桥实现 PCI 总线 ISA 接口的转换，以兼容以前的 ISA 设备。不过，连接在 ISA 总线上的设备将不具备 PCI 设备的地址映射功能。

使用 PCI 总线的微机的系统结构如图 12-1 所示。

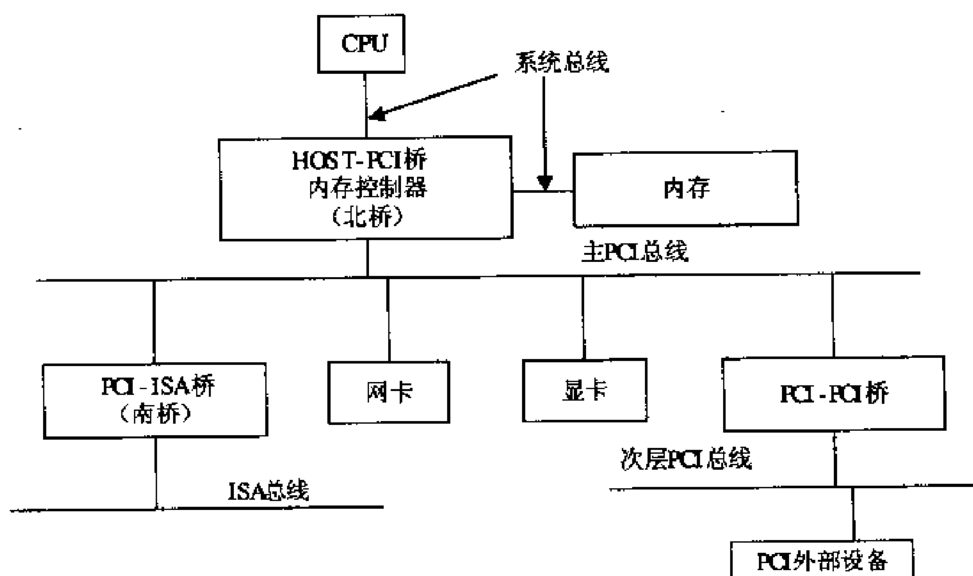


图 12-1　使用 PCI 总线的系统结构

从图 12-1 中可以看出，如果把 CPU 比作一个城市的中心，系统总线就是环绕城市的“一环路”，主 PCI 总线就好像是“二环路”，而 ISA 总线以及连在主 PCI 总线上的其他 PCI 总线就是“三环路”了。整个系统就是这样一层一层地向外扩展，而且 PCI 规范还采取了各种措施，以使 CPU 对外部 PCI 设备的存储器和 I/O 的访问就如直接访问系统总线上的存储器和 I/O。至于它如何实现这一点，将在 12.1.2 小节“PCI 配置空间”中说明。

由于 PCI 协议非常复杂，通常采用两种方式来实现它：一是使用专用芯片（如 AMCC 公司的 S593X 系列；PLX 公司的 PCI9052、PCI9054、PCI9050 等，放置于系统或 PCI 与 PCI 总线之间，提供控制和数据信号的接口电路）；二是使用 CPLD（如 ALTERA 公司的 FLEX8000）和 FPGA（如 Xilinux 的 XC3100A 等），它们不受插卡功能的限制。使用 PCI 专用接口芯片比较简单方便，设计者不必在处理系统与 PCI 总线接口上花费太多时间，所以一般采用这种方式。



12.1.2 PCI 配置空间

1. 简介

PCI 结构被设计用来替代 ISA 标准，主要出于以下目的：在计算机和其外围之间传送数据时有更高的性能，尽可能地做到平台无关性，使在系统中增减外围设备得到简化，第一个目的的实现在前面已经介绍了。平台无关性一直是计算机总线的一个设计目标，这是 PCI 的一个重要特征。不过对驱动程序开发者来说，最要紧的是对接口板自动检测的支持。PCI 设备是无跳线的（与大多数 ISA 外围不同），并且在引导时被自动配置。因此，设备驱动程序必须能够访问设备上的配置信息来完成初始化。

由于 PCI 是无跳线的，不能通过跳线告知系统 PCI 设备的信息，而且它又要实现在系统初始化时对其进行自动配置，那它是如何做到这一点的呢？答案就是 PCI 配置空间。除了 HOST—PCI 桥外，每一个 PCI 设备（包括 PCI—PCI 桥、PCI—ISA 桥）都要实现 PCI 配置空间，该配置空间是由一系列地址连续的配置寄存器组成的，在 PCI 配置空间中对 PCI 设备信息进行描述，包括设备的厂商、类别，设备所申请的存储器和 I/O 空间，设备的中断信息及 PCI 设备工作方式等。接下来，将对 PCI 配置空间加以说明。

整个配置空间共 256 个字节，其中会直接影响 PCI 设备特性的寄存器集中在 PCI 配置空间的前 16 个双字里，这个区域就是 PCI 配置头，目前 PCI2.2 规范定义了 3 种配置头的格式，分别是类型 0、1 和 2 型配置头。

- 类型 0：标准的 PCI 应用设备，除类型 1 和类型 2 外的所有 PCI 设备。
- 类型 1：PCI—PCI 桥设备，用于对 PCI 总线扩展，将两条 PCI 总线进行连接。
- 类型 2：PCI—CarBus（主要用于笔记本的插卡式总线）桥，在 PC Card 规范中对其进行了定义。

在这里只对类型 0 的配置头加以介绍，它的配置空间如图 12-2 所示。

2. 必须实现的寄存器

以下描述的寄存器在所有的 PCI 设备中都应当实现，包括桥设备。

首先是厂商号（Vender ID）、设备号（Device ID）、版本号（Revision ID）、类别号（Class ID）、子系统厂商号（Subvendor ID）和子系统号（Subsystem ID）。这些寄存器标明 PCI 设备是什么设备，启动时，系统将根据这些寄存器辨别出 PCI 设备，并为它加载相应的驱动程序。

（1）厂商 ID 配置寄存器。

该 16 位寄存器代表 PCI 设备制造商信息。本寄存器是只读寄存器，其中烧入的内容是由 PCI SIG 分配给该制造商的一个编号。PCI SIG 是一个 PCI 规范管理机构，每个生产 PCI 设备的厂商都应该向它申请一个自己的厂商 ID 号，每一个 PCI 厂商的 ID 都是惟一的，并且不可能是 0xffff。在系统启动时，会扫描 PCI 总线，在每一个 PCI 地址上读取厂商 ID 号，如果读到的数值为 0xffff，标明这个位置没有插入 PCI 卡；否则，就标明这个位置有 PCI 设备，同时，将读取该厂商的 ID 号。



Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Head Type	Latency Timer	Cache Line Size	0ch
Base Address Registers				10h 14h 18h 1ch 20h 24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2ch
Expenssion ROM Base Address				30h
Reserved				34h
Reserved				38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	3ch

图 12-2　类型 0 的 PCI 配置空间

（2）设备 ID 配置寄存器。

这 16 位寄存器由设备制造商定义，标明设备的用途。它与厂商 ID 配置寄存器一道向系统提供一个确定设备所需驱动程序的途径。

（3）系统厂商 ID 寄存器和子系统 ID 寄存器。

同样地，子系统厂商 ID 也是由 PCI SIG 进行管理，厂商从它那里获得一个惟一的标志。而子系统 ID 则是由厂商自定的。

这对寄存器的作用是显而易见的，如果使用了相同的 PCI 接口芯片，且供应商将厂商 ID 和设备 ID 已经进行硬件设置了，那么两个不同功能的设备必须有一种被区分开来的方法。利用不同的寄存器，系统就可以区分使用相同接口芯片的不同设备了。

（4）版本 ID 寄存器。

这个寄存器为 8 位，表示 PCI 设备的版本号。

（5）类别代码寄存器。

这是个 24 位的只读寄存器，共分为 3 个独立的 8 位单元：基本类型字节、子类型字节和编程接口字节。其中最高 8 位代表基本类型；中间 8 位代表子类型字节；最低 8 位代表



编程接口字节。它们分别代表设备的基本功能（例如大容量存储控制器）、细化的设备子类型（例如 IDE 大容量存储控制器）以及在一些情况下存储器指定的编程接口（例如 IDE 寄存器组的指定格式）。

当然，对于许多基本类型和子类型的组合来说，最低 8 位的编程接口一般都不用，硬连线为 0（即没有意义）。但是对于某些设备，比如 VGA 兼容设备和 IDE 控制器，编程接口字节还是有意义的。

类别代码寄存器的作用体现在当查找新的硬件的时候，会根据类别代码来自动判断该设备属于何种设备。然后操作系统根据类别代码自动选择驱动程序。

设备类别代码非常多，在这里就不详细一一说明了，具体情况请参看 PCI 2.2 规范的配置空间部分。

其次是命令寄存器（Command）和状态寄存器（Status），它们标明设备的工作方式、设备的能力和设备工作的状态。

（6）命令寄存器。

该寄存器提供了对 PCI 设备的访问方式以及 PCI 设备的工作模式。这是一个 16 位的寄存器，目前只有低端的 10 位有意义，高 6 位目前保留，命令寄存器各位的意义如图 12-3 所示。

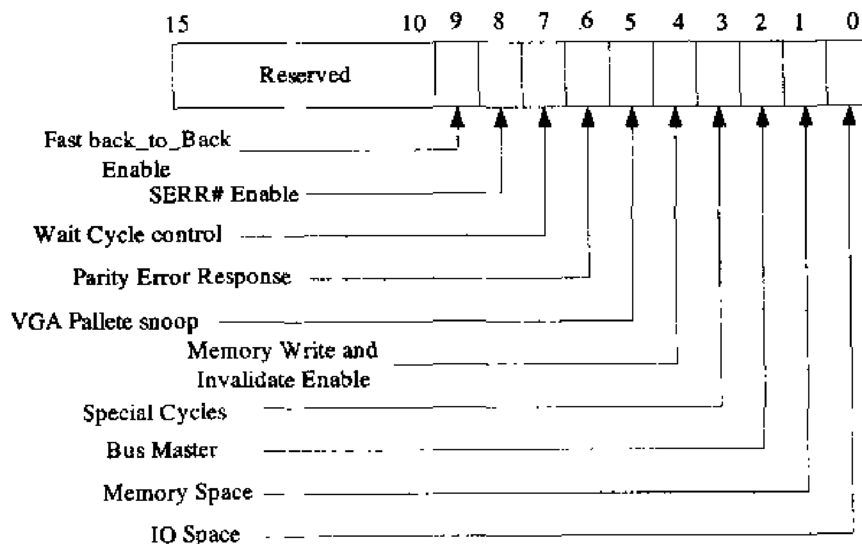


图 12-3　命令寄存器布局

命令寄存器各位含义的解释如下：

- I/O 空间位（I/O Space）：为 1 时表示 PCI 总线上出现的地址将被译码成 I/O 地址，0 则表示禁止，默认为 0。
- 存储器空间位（Memory Space）：为 1 时表示 PCI 总线上出现的地址将被译码成存储器地址，0 则禁止，（显然 0 位和 1 位只能有一个为 1，同时，在系统上电时，这两位都被设为 0，以免发生设备尚未配置好就被寻址的情况）。
- 总线主设备位（Bus Master）：为 1 表示该设备可以为主设备（前提是该设备具有主设备能力，如能够发起传送），为 0 表示该设备不能成为主设备，默认为 0。



- 特殊周期位（Special Cycles）：为 1 表示设备将会监视 PCI 总线的特殊周期（前提是设备具有监视总线特殊周期的能力），0 将会使设备忽略所有 PCI 特殊周期，默认为 0。
- 存储器写和使失效功能位（Memory Write and Invalidate Enable）：为 1 表示设备将会产生存储器写和使失效命令，当设置为 0 时，设备使用存储器写命令来代替。
- VGA 调色板检测位（VGA Palette Snoop）：本位只用于显示设备，为 1 将会使 VGA 兼容设备检测所有对 VGA 颜色调色板寄存器的写操作。对于非 VGA 设备，复位后该位为 1，而 VGA 兼容显示设备控制器复位后该位将被置为 0。
- 奇偶校验错响应位（Parity Error Response）：为 1 时设备将会通过使 PERR＃信号置为有效来向系统报告奇偶校验错误。为 0 的时候，奇偶校验错将不会在 PERR＃信号上显示出来。不过，无论如何，状态寄存器的奇偶校验错位还是需要的。该位默认置为 0。
- 步进控制位（Wait Cycle Response）：该位控制设备是否可以使用地址/数据步进。不使用步进功能的设备必须将该位置硬件为 0。总是使用步进功能的设备可以硬连线为 1。如果设备可以选择是否使用步进功能，则该位可以设定为可以读写的，同时该位复位后置为 1（步进是实现突发传送的关键，若 PCI 设备的数据传输量大，则应该使用步进）。
- SERR＃使能信号位（SERR# Enable）：为 1 表示设备能够驱动 SERR＃信号，利用它报告系统错误，为 0 表示不驱动。
- 快速背靠背使能位（Fast Back-to-Back Enable）：如果 PCI 总线主设备可以与不同的目标设备在相邻的两次传输中执行快速背靠背传输，那么这一位将用于使能或者禁止这一功能。当然，实现这一功能的前提是必须得到主设备和目标设备的支持。所以如果一个主设备的所有从设备都支持快速背靠背传输，那么应当把这一位置为 1，默认为 0。

（7）状态寄存器。

该寄存器用于记录一个 PCI 设备目前的工作状态。在状态寄存器中如果标明了具有某种功能，那么在硬件中必须有对这种功能的实现。

状态寄存器可读，并且读的方法与正常读没有差别。但是写的方法与正常写有差别，它只能对某些位进行重置操作。如果要重置某些位，只需将该位置为 1 即可。例如，要重置 14 位而不影响其他位，可以向状态寄存器写入 0x4000h。状态寄存器各位的含义如图 12-4 所示。

状态寄存器各位含义的解释如下：

- 0~4：只读，保留未用。
- 5：只读，66MHz 支持能力位。当置为 1 时表示设备工作于 66MHz；当置为 0 时表示设备不支持 66MHz 传送。
- 6：只读，用户自定义特征位（User Definition Feature，UDF）。当置为 1 时表示有用户自定义特征；当置为 0 时表示没有用户自定义特征。
- 7：只读，快速背靠背传送支持位。当置为 1 时表示此设备支持快速背靠背传输；



当置为 0 时表示不支持。

- 8：读写，主设备奇偶校验错位。此位只有主设备设置，并且只有符合下列 3 个条件才设置：
  - ➢ 总线主设备设置了 PERR#信号或者检测到 PERR#信号；
  - ➢ 总线设备是总线拥有者，并且是传送发起者；
  - ➢ 在它的命令寄存器中奇偶校验响应位已经置为 1。
- 9~10：只读，设备选择定时位。在选定设备时，定义 DEVSEL#的快慢。00b＝快速；01b＝中速；10b＝慢速；11b＝保留。
- 11：读写，发出目标设备终止位。一旦目标设备在传送中终止处理，那么目标设备在自身的状态寄存器中设置这个位。
- 12：读写，收到目标设备终止位。总线主设备收到目标设备终止信号后设置该位。
- 13：读写，收到主设备中止。
- 14：读写，发出系统错误。
- 15：发现奇偶校验错。

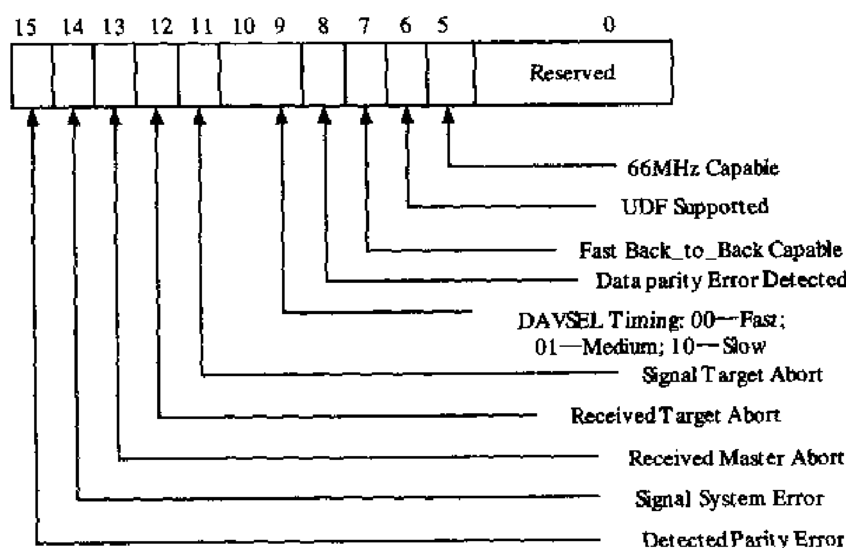


图 12-4 状态寄存器布局图

（8）首部类型寄存器。

这个单字节寄存器的 0~6 位定义了首部寄存器第 4~15 个双字的格式，位 7 定义了该设备是否属于多功能设备（位 7＝1）。

3. 其他配置寄存器

以下将要描述的配置寄存器都属于首部类型为 0 的 PCI 设备。这些寄存器是可选的，不必在每个 PCI 设备中都实现，但是对于具体的一个 PCI 设备，这其中有些寄存器是必须实现的。

- Cache 行容量寄存器（Cache Line Size）



对于使用存储器写和使失效命令的主设备而言，这个寄存器是必须实现的。这个寄存器中的值也被主设备用来决定是否用读、行读或多行读来访问内存。

这个可读写配置寄存器指明系统 Cache 行容量的双字递增数目。总线主设备为了保证在 Cache 行边界开始一次处理，必须知道 Cache 行的容量。如果这个寄存器的值为 0，总线主设备不可以使用存储器写与使失效命令，而只能使用写命令。

- 延迟定时器（Latency Timer）

对于需要执行突发传送操作的总线主设备而言，这个寄存器是必需的。延迟定时器定义了一个时间段，它表明当主设备启动一个传送时，它将在这个延迟定时器超时以前一直保持有总线的控制权。在启动后，延迟定时器将会在每个 PCI 时钟的上升沿自动减 1。

主设备一直持有总线直到以下任一情况发生：

> 主设备已经完成了该次传输。

> 目标设备发出了 STOP#信号，提前终止传输。

> 计时器超时，同时已经有其他 PCI 主设备抢占了总线控制权。

如果总线主设备进行传输的时候需要执行多于两个数据周期的突发，那么延迟定时器就一定要实现。否则，就可以把它硬连线为 0。延迟定时器中，低 3 位已经硬连线为 0 了，只有高 5 位可编程。这样用户是以 8 个 PCI 周期为单位分配时间片。复位后，该寄存器为 0。

- 基地址寄存器（Base Address Registers）

基地址寄存器在 PCI 设备功能实现上相当重要，前面说到 PCI 总线的一大优势就是系统自动进行资源配置，这个寄存器也起了很大的作用，它也是大多数 PCI 设备需要实现的寄存器。基地址寄存器的位置在配置空间的第 4 个双字一直到第 9 个，它们被用来存放 PCI 设备映射的内存地址或者使用的 I/O 空间的首地址。

PCI 规范使用一种机制，使 I/O 和存储器能够区分开来，即在基地址寄存器的最低位上，如果是 0，表明这个基地址寄存器指向的是一个存储器空间；如果是 1，那么就是指向一个 I/O 空间。它们的寄存器各位如图 12-5 所示。

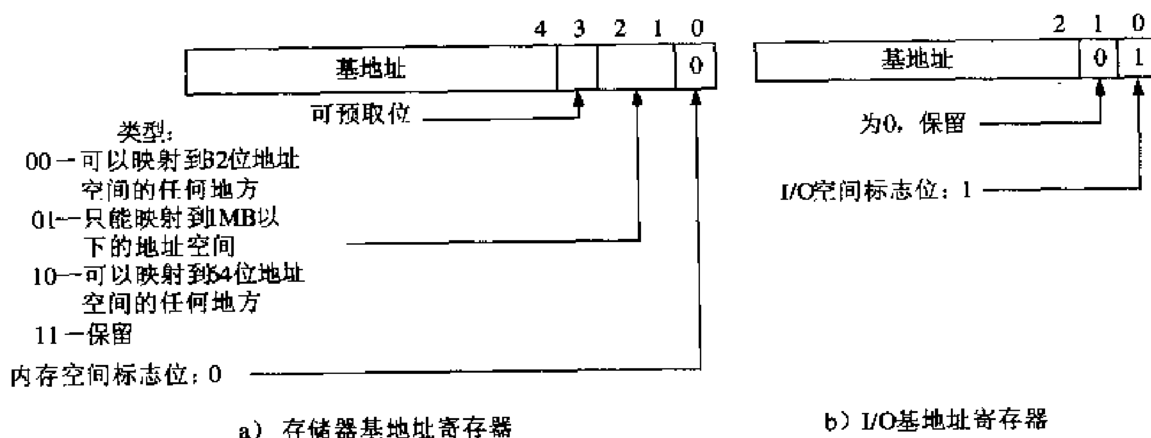


图 12-5 存储器和 I/O 基址寄存器

另外，就是如何知道存储器或者 I/O 空间的大小。PCI 是通过向一个基地址寄存器写全 1，然后读回这个寄存器的值，没有被改变的最低位的长度也就是需要向系统申请的地址空



间的大小。例如向基地址寄存器 0 写入 0xffffffff，如果读出 0x01100000，那么申请的地址空间为 1MB。

- 中断引脚寄存器（Interrupt Pin）

每个 PCI 设备总共有 4 条可以选择的中断请求引脚，它们分别是 INTA#、INTB#、INTC#和 INTD#。这个寄存器的值 01h~04h 分别表示设备使用 INTA#~INTD#的引脚。如果为 0 表示设备没有使用中断。

- 中断线寄存器（Interrupt Line）

可读写的中断寄存器用来表示 PCI 设备的中断是连到主机的中断控制器的哪一个引脚上。在 PC 机上，这个值为 00h~0fh，而 ffh 表示这个设备中断线未知，其他值保留。加电初始化时这个寄存器的值设为 0xffh。通过读取这个寄存器的值，驱动程序就可以注册中断处理函数了。

- Min_Lat 寄存器（Min_Lat）

这个寄存器是主设备专用的，同时也是可选的。其值的含义是主设备在启动一次数据传送任务后需要保持多久的总线所有权，进行计数的单位是 1/4 毫秒，0 表示对这一指标没有要求。

- Max_Lat 寄存器（Max_Lat）

这个寄存器也是主设备专用的，也可选。这个寄存器的值的含义是要求设备以多快的速度访问 PCI 总线，计数单位也是 1/4 毫秒。

至此，已把 PCI 配置空间的相关知识进行了详细的介绍，若有不清楚的地方，请参看 PCI 2.2 规范。

12.2 嵌入式 Linux 对 PCI 设备的支持

接下来，将介绍在嵌入式 Linux 中如何扫描 PCI 设备、如何为 PCI 设备分配所需资源和如何访问 PCI 配置空间等相关知识。

在具有 BIOS 的微机系统中，发现 PCI 设备、读取设备信息和分配设备所需资源都是由 BIOS 实现的，这一切工作在系统上电初始化时就已完成。但是在嵌入式系统中，许多时候系统是不带 BIOS 的，这时以上工作就需要由嵌入式操作系统来实现，在嵌入式 Linux 操作系统中提供了这些功能。接下来，就讨论在嵌入式 Linux 中如何实现对 PCI 设备的支持。

12.2.1 扫描 PCI 设备

前面曾介绍过，PCI 设备资源的分配是动态的，而且是自动的。要让嵌入式 Linux 系统在开机时能够自动给 PCI 设备分配资源，首先，系统得知道在哪个位置上有 PCI 设备，哪个位置上没有设备，这就需要通过扫描得到相关信息。

在介绍配置空间的时候曾介绍过，操作系统是通过读取配置空间寄存器中的厂商 ID 来



判断在某个位置是否有 PCI 设备的存在。但问题是操作系统又是如何知道配置空间的位置呢？假设为每种设备的配置空间分配不同的 I/O 空间，试想一下，如果每种设备都需要保留 256 字节的配置寄存器组，那么有 1024 种设备就得保留 256KB 地址空间，但是谁又知道到底有多少种 PCI 设备呢？所以这种方式是不可取的，比较好的办法就是让所有设备的配置寄存器组都采用相同的地址，由所在总线上的 PCI 桥在访问时使用其他条件来区分。而 CPU 则通过一个统一的入口地址向“HOST-PCI 桥”发出命令，由相应的 PCI 桥间接地完成具体的读写。

例如，对于 x86 结构处理器，PCI 总线的设计者在 I/O 地址空间保留了 8 个字节用于访问 PCI 设备的配置空间。那就是 0xCF8～0xCFF，这 8 个字节实际上构成两个 32 位的寄存器，第一个是地址寄存器 0xCF8，第二个是数据寄存器 0xCFF。要访问某个设备的某个配置寄存器时，CPU 先往地址寄存器写入目标地址，然后通过数据寄存器读写数据。不过写入地址寄存器中的目标地址不再是传统意义上的 32 位地址，而是一种包括总线号、设备号、功能号以及配置寄存器地址的合成地址，它的组成如图 12-6 所示。

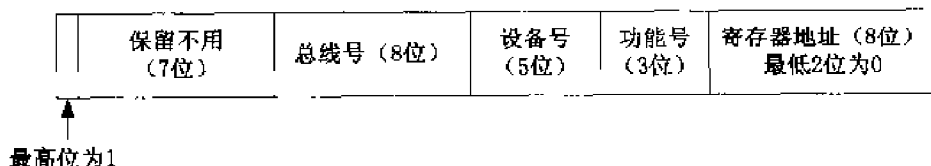


图 12-6　写入地址寄存器 0xCF8 的综合地址

这里的总线号、设备号以及功能号是对配置寄存器地址的扩充，用于查找具体设备。首先每条 PCI 总线都有其总线号，与 HOST-PCI 桥相连的 PCI 总线的总线号固定为 0，其余的则由系统在初始化扫描过程中指定，每扫描到一条 PCI 总线便为其定义一条总线号，序号依次递增。设备号一般代表着一块 PCI 接口卡（更确切地说应当是一块 PCI 接口芯片），通常取决于插槽的位置。每块 PCI 接口卡上又可以有若干个功能模块，这些功能模块共用一个 PCI 接口芯片，包括其中用于地址映射的电子线路，以降低成本。从逻辑上说，每个功能其实就是一项设备，所以功能号和设备号加在一起又称为“逻辑设备号”。每个 PCI 卡上最多可以容纳 8 个逻辑设备。

事实上还有一个问题，就是在尚未为各条 PCI 总线定义总线号之前，CPU 又如何访问特定的总线呢？事实上，一开始的时候，只有 0 号 PCI 总线是可以访问的。在扫描 0 号总线时如果发现上面某个设备是 PCI 桥，就为之指定一个新的 PCI 总线号，例如 1，这样 1 号总线就可以访问了……依次扫描，就可以访问所有总线了。

在 PC 机中，BIOS 提供了对 PCI 总线操作的支持，即 PCI 总线和设备的扫描以及配置所需的功能和服务。早期的 Linux 内核也依靠 BIOS 完成对 PCI 的扫描和配置操作，但是后来已经实现了自己的 PCI 总线操作而不依赖于 BIOS。现在，是否采用 PCI BIOS 是一个条件编译，在内核代码中可以指定，在不支持 PCI BIOS 的嵌入式应用中，一般都由嵌入式 Linux 操作系统自己实现 PCI 操作。

这里的重点是嵌入式 Linux 自己对 PCI 设备的扫描操作，首先是发现“HOST-PCI 桥”，

并且在系统 I/O 中请求分配 0xCF8~0xCFF 的 I/O 空间，这是由 pci_check_direct()函数实现的。这个函数的代码在 uClinux 源代码部分 linux/linux-2.4.x/arch/i386/kernel/pci-pc.c 中。它的代码如下：

```
static struct pci_ops * __devinit pci_check_direct(void)
{
        unsigned int tmp;
        unsigned long flags;

        __save_flags(flags);
        __cli();
        /*
         * Check if configuration type 1 works.
         */
        if (pci_probe & PCI_PROBE_CONF1) {
                outb (0x01, 0xCFB);
                tmp = inl (0xCF8);
                outl (0x80000000, 0xCF8);
                if (inl (0xCF8) == 0x80000000 &&
                        pci_sanity_check(&pci_direct_conf1)) {
                        outl (tmp, 0xCF8);
                        __restore_flags(flags);
                        printk(KERN_INFO "PCI: Using configuration type 1\n");
                        request_region(0xCF8, 8, "PCI conf1");
                        return &pci_direct_conf1;
                }
                outl (tmp, 0xCF8);
        }
        /*
         * Check if configuration type 2 works.
         */
        if (pci_probe & PCI_PROBE_CONF2) {
                outb (0x00, 0xCFB);
                outb (0x00, 0xCF8);
                outb (0x00, 0xCFA);
                if (inb (0xCF8) == 0x00 && inb (0xCFA) == 0x00 &&
                        pci_sanity_check(&pci_direct_conf2)) {
                        __restore_flags(flags);
                        printk(KERN_INFO "PCI: Using configuration type 2\n");
                        request_region(0xCF8, 4, "PCI conf2");
                        return &pci_direct_conf2;
                }
```




```
        }
        __restore_flags(flags);
        return NULL;
}
```

如前所述，"HOST-PCI 桥" 的 I/O 口地址 0xCF8~xCFB 为地址口，0xCFC~0xCFF 为数据口。但是也有可能系统中根本就没有 PCI 总线的存在；或者再进一步，没有 PCI 总线的存在，却凑巧有一个 ISA 设备正在使用这些地址。针对这些情况，PCI 总线标准规定了测试的方法。在这里先按 1 型操作尝试，如果成功就从 return &pci_direct_conf1 处返回；如不成功则按 2 型再尝试。注意这里的 1 型和 2 型不是指 PCI 头部类型，2 型是在 PCI 总线发展的早期的一种"宿主—PCI 桥"，现在已经不用了。如果都不成功就表明没有 PCI—HOST 桥。一般应该检测到 1 型成功，但是探测到一个 1 型 "HOST—PCI 桥" 后，要进一步调用 pci_sanity_check()再加以验证（通过读取 "HOST—PCI 桥" 的类别信息和厂商信息，如果都不满足预设的要求，那么也就说明先前检测到的 "PCI—HOST 桥" 是不存在的）。

接下来，就是扫描总线了，这是由 linux/linux-2.4.x /drivers/pci/pci.c 中的 pci_scan_bus() 函数实现的，它的代码为：

```
struct pci_bus * __devinit pci_scan_bus(int bus, struct pci_ops *ops, void *sysdata)
{
        struct pci_bus *b = pci_alloc_primary_bus(bus);
        if (b) {
                b->sysdata = sysdata;
                b->ops = ops;
                b->subordinate = pci_do_scan_bus(b);
        }
        return b;
}
```

这里有一个重要的数据结构 struct pci_bus，这种结构定义在 linux/linux-2.4.x include/linux/pci.h 中，每一条 PCI 总线都由一个这样的数据结构来表达。系统中每条 PCI 总线都有一个编号，主总线编号为 0。所有的 pci_bus 数据结构都互相连接在一起，形成一棵 PCI 总线树，树的根是一个代表 "HOST—PCI 桥" 的 pci_bus 结构。内核中有一个队列头 pci_root_buses，所有代表着"HOST—PCI 桥"的 pci_bus 结构都通过其内部的队列头 node 挂在这个队列中。同时，每个 pci_bus 结构本身又维持着两个队列，一个是 devices，凡是连接在这条总线上的 PCI 设备都有一个 pci_dev 数据结构挂在这个队列中；另一个是 children，凡是通过 "PCI—PCI 桥" 连接在这条总线上的下层 PCI 总线都有一个 pci_bus 的数据结构挂在这个队列中。这样，从队列 pci_root_buses 开始的整个层次结构就反映了系统中 PCI 总线和设备的连接和配置情况，它们的数据结构层次如图 12-7 所示。

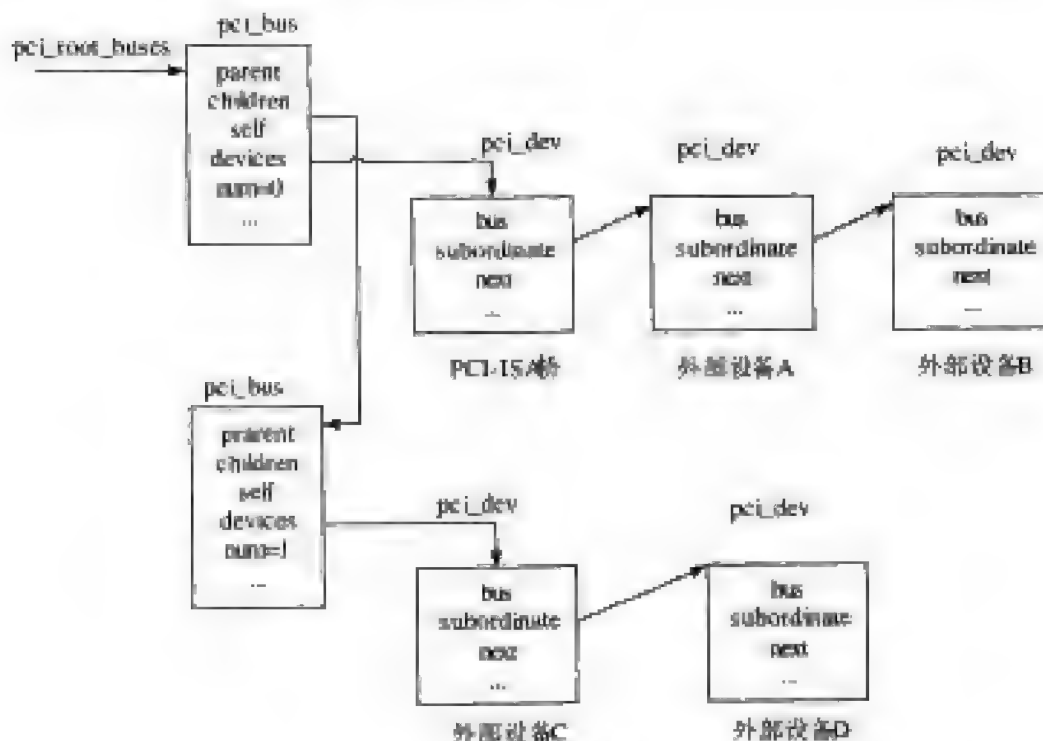


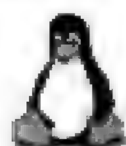
图 12-7　PCI 数据结构

pci_scan_bus()函数的目的正是要在内存中建立起这样一个层次结构，它首先调用 pci_alloc_primary_bus()函数为主总线分配数据结构，并把它挂入总线队列，然后调用 pci_do_scan_bus()函数进行总线扫描。pci_do_scan_bus()函数在 linux/linux-2.4.x drivers/pci/pci.c 中实现，其代码为：

```
unsigned int __devinit pci_do_scan_bus(struct pci_bus *bus)
{
        unsigned int devfn, max, pass;
        struct list_head *ln;
        struct pci_dev *dev, dev0;

        DBG("Scanning bus %02x\n", bus->number);
        max = bus->secondary;

        /* Create a device template */
        memset(&dev0, 0, sizeof(dev0));
        dev0.bus = bus;
        dev0.sysdata = bus->sysdata;
        /* Go find them, Rover! */
        for (devfn = 0; devfn < 0x100; devfn += 8) {
```




```
			dev0.devfn = devfn;
			pci_scan_slot(&dev0);
	}
	/*
	 * After performing arch-dependent fixup of the bus, look behind
	 * all PCI-to-PCI bridges on this bus.
	 */
	DBG("Fixups for bus %02x\n", bus->number);
	pcibios_fixup_bus(bus);
	for (pass=0; pass < 2; pass++)
			for (ln=bus->devices.next; ln != &bus->devices; ln=ln->next) {
			dev = pci_dev_b(ln);
			if (dev->hdr_type == PCI_HEADER_TYPE_BRIDGE || dev->hdr_type ==
PCI_HEADER_TYPE_CARDBUS)
				max = pci_scan_bridge(bus, dev, max, pass);
		}
	/*
	 * We've scanned the bus and so we know all about what's on
	 * any devices.
	 *
	 * Return how far we've got finding sub-buses.
	 */
	DBG("Bus scan for %02x returning with max=%02x\n", bus->number, max);
	return max;
}
```

扫描一条 PCI 总线的直接目的就是逐个地发现连接在总线上的 PCI 设备，为其建立起 pci_dev 数据结构并挂入相应的队列。所以在这里先准备好下一个空白的 pci_dev 结构，然后依次对各个 PCI 接口通过 pci_scan_slot()扫描，每次扫描 8 个功能，即 8 个逻辑设备（这是每块 PCI 接口卡上的最大容量），只要扫描到新设备，就对这个结构进行填充。在 pci_scan_slot()中通过读取配置空间头部信息，判断逻辑设备是否存在，并把逻辑设备链表返回。如果设备存在，下一步就是为它们分配资源了。

12.2.2 为 PCI 设备分配资源

在介绍如何实现 PCI 设备分配资源之前，首先介绍一个代表 PCI 设备的 struct pci_dev 数据结构，它定义在 uClinux 源代码部分 linux/linux-2.4.x /include/linux/pci.h 中，此数据结构在以后的 PCI 驱动中也会用到，所以在此对其进行详细说明。该数据结构的实现代码如下：

```
struct pci_dev {
    struct list_head global_list;            /*系统中所有 PCI 设备结构的头指针*/
    struct list_head bus_list;               /*每个 PCI 总线中的设备头指针 1 */
    struct pci_bus    *bus;                  /*设备所在的总线*/
    struct pci_bus    *subordinate;          /*设备所在总线的下一级总线*/

    void        *sysdata;                    /*系统扩展用的数据*/
    struct proc_dir_entry *procent;          /*设备在/proc/bus/pci 中的入口*/

    unsigned int       devfn;                /*设备和功能编码*/
    unsigned short     vendor;
    unsigned short     device;
    unsigned short     subsystem_vendor;
    unsigned short     subsystem_device;
    unsigned int       class;                /*3 字节：表示 base.sub.prog-if*/
    u8          hdr_type;                    /*PCI 头类型*/
    u8          rom_base_reg;                /*哪个配置寄存器控制 ROM*/

    struct pci_driver *driver;               /*设备驱动程序*/
    void        *driver_data;                /*驱动程序参数*/
    u64         dma_mask;                    /*设备实现地址总线的掩码，一般为
                                               0xffffffff*/

    u32                current_state;        /*当前工作状态*/

    /*以下两个域描述与 ID 相匹配的设备*/
    unsigned short vendor_compatible[DEVICE_COUNT_COMPATIBLE];
    unsigned short device_compatible[DEVICE_COUNT_COMPATIBLE];
    /* 用以下各域代替中断线和基址寄存器保存相关数据 */
    unsigned int    irq;
    struct resource resource[DEVICE_COUNT_RESOURCE]; /*I/O 和存储器以及扩展的
                                                       ROM 的区域*/

    struct resource dma_resource[DEVICE_COUNT_DMA];
    struct resource irq_resource[DEVICE_COUNT_IRQ];

    char      name[90];                      /*设备名称*/
    char      slot_name[8];                  /*插槽名称*/
    int       active;                        /*对于 ISAPnP 表示设备激活*/
    int       ro;                            /*对于 ISAPnP 表示只读*/
    unsigned short    regs;                  /*对于 ISAPnP 表示支持的寄存器*/
```




```
        int (*prepare)(struct pci_dev *dev);    /*ISAPnP 钩子*/
        int (*activate)(struct pci_dev *dev);
        int (*deactivate)(struct pci_dev *dev);
};
```

在初始化扫描到 PCI 设备后，就要读取它的设备信息，把这些信息写到一个 pci_dev 数据结构中去。具体过程如下：

（1）在 pci_do_scan_bus()中调用 pci_scan_slot()对每个逻辑设备进行扫描，从而获取头部类型信息，再在 pci_scan_slot()中调用 pci_scan_dev()读取具体逻辑设备的厂商信息。前面讲过，逻辑设备的配置寄存器中有些信息是由厂商提供，并且是固化在里面的，有些信息（如地址映射和总线号）则有待设置。

（2）在 pci_scan_dev()中再调用 pci_setup_device()进一步把设备信息填入 pci_dev 数据结构中去，它先读入用于设备类别和版本号的长字，然后根据具体设备的头部类型进行处理。对于一般的 PCI 设备，它通过 pci_read_irq(dev)读入并设置中断信息，再使用 pci_read_Base()读取基地址信息放到 pci_dev 结构中。

（3）在这里先说明嵌入式 Linux 对中断的处理。PCI 设备通常都可以发出中断请求，所以在配置寄存器中有两个字节 PCI_INTERRUPT_PIN 和 PCI_INTERRUPT_LINE，从而反映该设备的中断请求线与总线及系统中断处理器的连接方式。

（4）在总线上 INTA~INTD 共有 4 条中断请求线，从而在 PCI 插槽上有 4 根"针"。并非所有的 PCI 设备都能产生中断请求，如果 PCI_INTERRUPT_PIN 字节为 0，就表示本地设备不能产生中断请求。但是，如果一个 PCI 设备能够产生中断请求，那么在设备内部必定已经把中断请求连到 PCI 总线的某条中断请求线上。此时 PCI_INTERRUPT_PIN 字节的数值（1～4）就表示该设备的中断请求线连在哪一条线上。这种连线是由硬件决定的，所以 PCI_INTERRUPT_PIN 字节是一个只读的字节，不能通过软件设置。

可是，连在哪一条 PCI 中断请求线上只完成中断请求的一半，还有一个最终连接到系统的中断控制器（8259A 或 APIC）上的哪条中断请求线上的问题，通常称之为"中断请求路径"。这是由软件选择和设置的，选择的结果存储在 PCI_INTERRUPT_LINE 字节中。需要指出的是，PCI_INTERRUPT_LINE 的目的只是保存信息，而并不带有控制功能。所以如果把这个字节的内容从 8 改为 10 并不意味着改变了连接的目标。这里通过 pci_read_irq()读入这两个关于中断的字节，并把它们记录在 pci_dev 中。

关于 PCI 设备中断的另一大特征是中断共享，ISA 卡的一个重要局限在于中断是独占的（计算机的中断号只有 16 个，系统又用掉了一些），这样当有多块 ISA 卡要用中断时就会有问题了。PCI 总线则可以实现多个 PCI 设备共用一个中断号，中断共享的实现由硬件与软件两部分组成。

硬件上，采用电平触发的方法，中断信号在系统一侧用电阻接高，而要产生中断的板卡上利用三极管的集电极将信号拉低。这样不管有几块板产生中断，中断信号都是低；而只有当所有板卡的中断都得到处理后，中断信号才会回复高电平。其实现原理如图 12-8 所示。

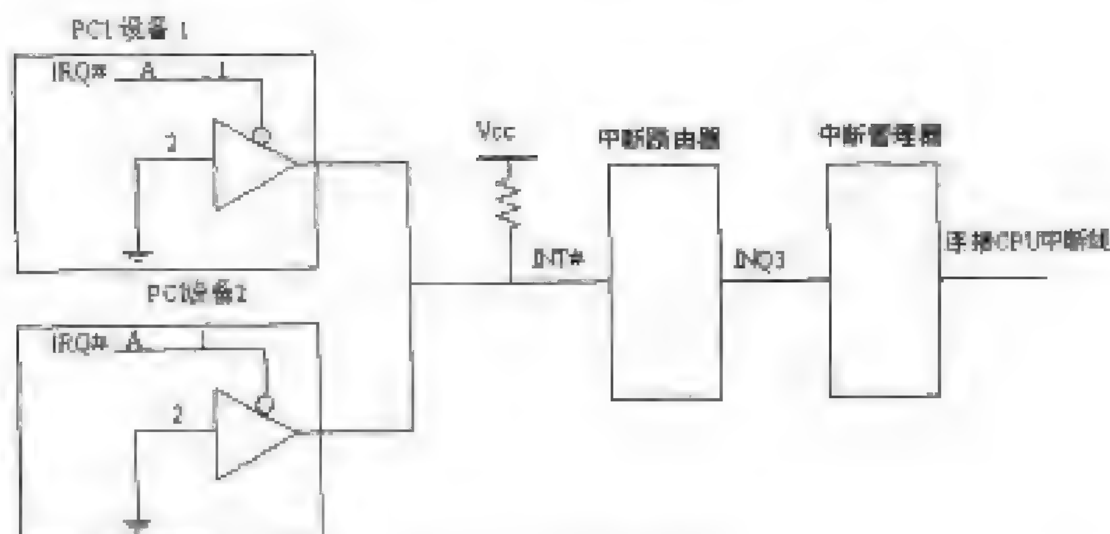


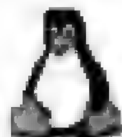
图 12-8　中断共享硬件原理

软件上，采用中断链的方法：假设系统启动时，发现板卡 A 用了中断 3，就会将中断 3 对应的内存区指向板卡 A 对应的中断服务程序入口 ISR_A；当系统发现板卡 B 也采用中断 3 时，就会将中断 3 对应的内存区指向 ISR_B，同时将 ISR_B 的结束指向 ISR_A，以此类推，就会形成一个中断链。而当有中断发生时，系统跳转到中断 3 对应的内存，也就是 ISR_B，ISR_B 就要检查是不是 B 卡的中断，如果是，就加以处理，并将板卡上的拉低电路放开；如果不是，则呼叫 ISR_A，这样就完成了中断的共享。

另外，PCI 设备中一般都带有一些 RAM 和 ROM 区间，通常的控制/状态寄存器和数据寄存器也往往以 RAM 区间的形式出现，这些地址都要先映射到系统总线上，再映射到内核的内存空间。现在要通过 pci_read_bases()把这些区间的大小和当前的地址读进来。对于一般的 PCI 设备，最多有 6 个这样的 RAM 或 I/O 区间。

从配置寄存器组的一个双字中读出了区间的起始地址后，往同一双字中写入全 1，即 0xffffffff，接着再从同一地址中读，这时取得的数值便是区间的大小。这个数值的低 4 位（存储器空间）或低 3 位（I/O 空间）为控制信息，这一点与起始地址的格式相似，但是在其高 28 位或者高 29 位中只有位置最低的那位 1 才有效。区间的大小必定是 2 的某次幂，所以按理说读得的数值中应该只有一位为 1，其他各位都为 0。但在实际中却常常读得的多位是 1，此时只有位置最低的那个 1 有效，所以要通过 pci_size()加以换算，它的代码为：

```
static u32 pci_size(u32 base, unsigned long mask)
{
        u32 size = mask & base;         /*屏蔽掉低几位*/
        size= size & ~(size-1);         /*得到最低位表示的大小*/
        return size-1;                  /*extent=size-1*/
}
```



这里先用 mask 把最低的 4 位或 3 位屏蔽掉，然后再把位置最低的那个 1 抽取出来。例如，把最低的 4 位或 3 位屏蔽掉以后得到的 size 的数值是 0xffff0100，则(size-1)为 0xffff00ff，而~(size-1)就是 0x0000ff00。最后得到 size=0xffff0100&0x0000ff00=0x100。这样，就把 size 的二进制数值中位置最低的那个 1 抽取出来，从而得到 size 的实际数值为 0x100，即区间的大小为 256，返回 255。在这个实例中，就表示该区间在映射后的起始地址必须是 64KB 边界上的第一个或第二个 256 字节。

注意：读取了区间的大小后，还要把起始地址写回这个长字，恢复其原状。

除常规的 6 个存储区外，设备上还可以提供一个扩充的 ROM 空间。ROM 基地址寄存器用于指示 ROM 的起始地址和大小。对 ROM 空间地址的解读与 RAM 区间有所不同，其最低的 11 位都是保留的，其中 bit0 为 PCI_ROM_ADDRESS_ENABLE，为 1 时表示区间有效。

12.2.3 对 PCI 配置空间的访问

在驱动程序检测到设备后，它通常要对 3 个地址空间读或写：内存、I/O 端口和配置。其中访问配置空间对驱动程序来说极为重要，因为这是它发现设备被映射到内存和 I/O 空间具体位置的惟一的办法。

至于驱动程序，嵌入式 Linux 为它们提供了标准的访问接口，配置空间可以通过 8 位、16 位、32 位的数据访问接口来访问。相关函数的原型在< linux/linux-2.4.x include/linux/pci.h>中定义，例如读取配置空间的函数为：

```
int pcibios_read_config_byte(unsigned char bus, unsigned char function,          unsigned
char where, unsigned char ptr);              /*读取配置空间 8 位数据*/
int pcibios_read_config_word(unsigned char bus, unsigned char function,          unsigned
char where, unsigned char *ptr);             /*读取配置空间 16 位数据*/
int pcibios_read_config_dword(unsigned char bus, unsigned char function,         unsigned
char where, unsigned char *ptr);             /*读取配置空间 32 位数据*/
```

它们的功能是从由 bus 和 function 确定的设备的配置空间中分别读取 1、2、4 个字节。参数 where 是从配置空间开始处的以字节为单位的偏移。从配置空间取出的值通过 ptr 返回。这些函数的返回值是错误代码。字和双字函数会将从配置空间读出的值自动转换为处理器本身的字节序，因此开发者并不需要处理字节序。

不过，由于 Linux 内核以前也采用 PCI BIOS 提供的 PCI 服务，所以许多函数的函数名都带有前缀 pcibios。尽管现在已经可以不涉及 BIOS，却还是保留了其中一些函数的函数名而没有加以改变，从而避免引起混乱。

另外，就是对配置空间的写操作函数，这些函数的原形如下：

```
int pcibios_write_config_byte(unsigned char bus, unsigned char function,         unsigned
```

```
char where, unsigned char val);
int pcibios_write_config_word(unsigned char bus, unsigned char function,            unsigned
char where, unsigned short val);
int pcibios_write_config_dword(unsigned char bus, unsigned char function,           unsigned
char where, unsigned int val);
```

向配置空间里写 1、2、4 个字节，设备仍由 bus 和 function 确定，要写的值由 val 传递。字和双字函数在向外围设备写之前将数值转换为小头字节序。

访问配置变量的最好方法是使用在<include/linux/pci.h>中定义的符号名，例如：

```
#define PCI_BASE_ADDRESS_0      0x10/*0 号基地址位置*/
#define PCI_BASE_ADDRESS_1      0x14/*1 号基地址位置*/
...
```

如果已经知道一个描述 PCI 设备的数据结构 struct pci_dev *pdev，则上面的访问函数也可以用下面的函数来代替：

pci_read_config_byte(pdev, where, valp);...

pci_write_config_byte(pdev, where, val);...

下面的两行程序通过给 pcibios_read_config_byte 的 where 传递符号名来获取一个设备的修正版本 ID：

```
Unsigned char jail_get_revision(unsigned char bus, unsigned char fn)
{
  unsigned char *revision;
 pcibios_read_config_byte(bus,fn, PCI_REVISION_ID,&revision);
  return revision;
}
```

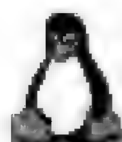
注意：当访问多字节值时，程序一定要记住字节序的问题。

12.3 编写嵌入式 Linux 下 PCI 驱动程序

在嵌入式 Linux 下扩展应用系统自己的 PCI 设备，最重要的就是要编写用户设备的驱动程序，驱动程序的编写包括发现设备、读取设备信息、申请资源、编写设备操作函数和注册设备操作等。接下来，将对这些内容进行说明。

12.3.1 编写 PCI 驱动程序

对于 PCI 驱动的编写，目前有两种架构，一种是设备驱动程序自己完成所有探测工作



的架构，包括探测系统是否支持 PCI 总线、要驱动的设备是否存在等；另一种是新型的驱动程序架构，它调用操作系统提供的 PCI 层，把多数设备的探测工作留给了 PCI 层，并且支持热插拔功能。接下来，将对这两种驱动程序架构加以说明。

1. 程序自己探测设备的驱动架构

程序自己探测设备的驱动架构的实现过程如下：

（1）判断系统中 PCI 总线是否存在，判断 PCI 总线是否存在的函数为 pci_present()，它的函数原形如下：

```
int pci_present(void);
```

本函数的返回值会告诉调用它的程序计算机是否支持 PCI。如果 BIOS 支持 PCI，那么返回值为真，否则为假。要知道对于内核而言，系统是否支持 PCI 是可选的，因此，在进行后续 PCI 操作之前，应先检查一下这个函数的返回值。

（2）如果系统支持 PCI 总线，下一步就是查找要驱动的 PCI 设备，在 PCI 配置头中有多个寄存器表示 PCI 设备信息，如厂商 ID、设备 ID 和类别 ID 等，只要在宏定义中对这些信息进行定义，就可以通过这些信息查找在系统中是否存在需要驱动的设备。

这类驱动架构主要调用以下函数查找设备：

- 通过生产商或设备 ID 号： pci_find_device()。

例如，使用下列代码查找设备：

```
struct pci_dev *dev = NULL;
while (dev = pci_find_device(VENDOR_ID, DEVICE_ID, dev))
configure_device(dev);
```

这个函数把查找到的设备用 pci_dev 的数据结构表示，如果没有查找到设备，就返回空指针。

- 通过种类 ID 号搜索：pci_find_class()。

如：

```
struct pci_dev *dev;
dev=pci_find_class(CLASS_ID, dev);
```

- 通过厂商/设备 ID 和子系统厂商/设备 ID 查找设备：pci_find_subsys()。

如：

```
struct pci_dev *dev;
pci_find_subsys(VENDOR_ID,DEVICE_ID,SUBSYS_VENDOR_ID, SUBSYS_DEVICE_
ID, dev);
```

- 有时候需要根据比较复杂的规则决定要查找的设备，这时就可以通过遍历来发现所有设备，根据每个设备的具体情况作出选择。

如：

```
    struct pci_dev *dev;
pci_for_each_dev(dev) {
  ... do anything you want with dev ...
}
```

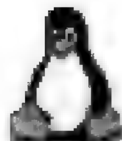
（3）如果找到了设备，就可以通过对设备配置空间的访问来初始化设备了。对设备配置空间的访问函数是由系统提供的，在前面已经介绍过了。

在程序中可以读取配置空间中的存储器、I/O 映射地址空间位置和大小，并对它们进行验证。另外如果 PCI 设备中使用了中断，还应该读取中断行号，安装中断处理程序。

一件值得高兴的事是启动初始化时为 PCI 设备处理了所有的资源配置工作。一般来说 PCI BIOS 具体做这些工作，但是在没有 BIOS 的平台上，这项工作由 Linux 内核初始化代码来做。到驱动程序查找到 PCI 卡的时候，它已经被分配了系统资源。只要找到 PCI 设备，它的资源信息即被分配并记录在 struct pci_dev 的数据结构中。

在查找到 PCI 设备后，就可以在 setup_device()函数中用下列代码进行初始化了：

```
    void setup_device(struct pci_dev *dev)
{
  int io_addr = dev->base_address[0] & PCI_BASE_ADDRESS_IO_MASK;
  int irq = dev->irq;
    u8 rev;
    pci_read_config_byte(dev, PCI_REVISION_ID, &rev); /*读取设备版本号*/
    /*打印设备信息*/
    if (rev<64)
    printk("Found a WonderWidget 500 at I/O 0x%04X, IRQ %d.\n",io_addr, irq);
    else
    printk("Found a WonderWidget 600 at I/O 0x%04X, IRQ %d.\n",io_addr, irq);
    /*检查是否被分配了中断*/
    if (irq==0)
    {
    printk(KERN_ERR "BIOS has not assigned the WonderWidget an interrupt.\n");
    return;
    }
    /*现在可以进行设备初始化了*/
    init_device(io_addr, irq, rev<64 ? 0 : 1); /*设备初始化函数*/
    /*如果设备是 PCI 主设备还要进行设定*/
    pci_set_master(dev);
}
```



在 PCI 设备被 BIOS 或者 Linux 内核代码配置后，某些特性可能会被屏蔽掉。如多数 BIOS 都会清掉 "master" 位，如果设备是主设备，则将导致板卡不能随意向主存中复制数据。在 Linux 2.2 中提供了一个辅助函数：

```
pci_set_master(struct pci_dev * dev);
```

这个函数会检查是否需要设置标志位，如果需要，则会将 "master" 位置位。

另外，在系统启动后，PCI 配置空间的命令寄存器中的存储器和 I/O 访问使能位是默认被置为 "否" 的。在 PCI 能够工作之前，应该把它们启用使能，其实现代码如下：

```
/*首先读出 PCI_COMMAND 的状态*/
pci_read_config_word(pdev,PCI_COMMAND,&cmd);
/*重新向 PCI_COMMAND 域写入使能命令，使能访问 PCI 设备的存储器和 I/O 地址空间
*/
pci_write_congfig_word(pdev,PCI_COMMAND,cmd|PCI_COMMAND_IO|PCI_COMMAN
D_MEMORY);
```

PCI 总线中一个重要的概念是共享中断处理，这在 ISA 总线设备中一般是看不到的。PCI 总线中断也是电平触发的（level-triggered），也就是说，中断一直在那里，直到设备去清除它。这些特性给驱动程序处理中断加上了一些重要的限制。

驱动程序在注册 PCI 中断时，总是应该带上 SA_SHIRQ 标志，用来指明中断线是可以共享的。如果不这样做，那么系统中使用这个中断的其他设备就有可能不能正常工作。

由于中断是共享的，PCI 设备驱动程序和内核都需要进行沟通，沟通的方法是调用函数 request_irq()进行中断注册。request_irq 的函数原型为：

```
int request_irq(unsigned int irq),
            void (*handler)(int, void *, struct pt_regs *),
            unsigned long irqflags,
            const char * devname,
            void *dev_id)
```

其中，irq 是申请的中断号；(*handler)是中断处理函数入口；irqflags 是中断标志，若为 SA_SHIRQ 表示这个中断号是共享的；devname 是申请中断的设备名。另外，必须用一个非空（non-NULL）的 dev_id 指针来注册共享中断，否则，当需要用 free_irq 来释放一个中断时，内核不能区分不同的中断处理例程。dev_id 将被送到中断处理例程，因此它非常重要。如果中断申请成功，函数返回非 0，否则返回 0。例如，可以用如下语句注册中断处理函数：

```
if (request_irq(dev->irq, dev_interrupt,
SA_SHIRQ, "wonderwidget",dev))
return -EAGAIN;
```

结束时，可用以下语句来正确释放中断：

```
free_irq(dev->irq, dev);
```

由于中断处理例程在被调用时已经收到 dev 参数，这就使事情变得很简单了，而不必搜寻使用该中断的设备，通常可以这样做：

```
static void dev_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
struct wonderwidget *dev = dev_id;
u32 status;
/*注意，如果判断出产生这个中断的不是自己的设备，
应该马上退出中断*/
if((status=inl(dev->port))==0)          /*不是自己设备的中断*/
return;
if(status&1)
handle_rx_intr(dev);                    /*是自己设备的中断，进行处理*/
…
}
```

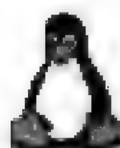
2. 新型的 PCI 驱动程序架构

新型的设备驱动程序在初始化的时候调用函数 pci_register_driver()，此函数的原型为：

```
int pci_register_driver(struct pci_driver *drv)
```

它的参数是一个 struct pci_driver 类型的指针。struct pci_driver 包括以下一些项：

```
struct pci_driver {
        struct list_head node;        /*链表结构头*/
        char *name;                   /*驱动的名字*/
        const struct pci_device_id *id_table;     /*指向驱动程序感兴趣的设备的 ID 列表，如
果设为 NULL 则感兴趣的设备是系统识别的所有设备*/
        int (*probe)  (struct pci_dev *dev, const struct pci_device_id *id);      /*一个探测函
数的指针，用来探测所有与 ID 表中匹配的设备。这个函数得到一个指向所有查找到的设
备的 pci_dev 指针。驱动程序接受设备的时候返回 0，否则返回一个错误代码*/
        void (*remove) (struct pci_dev *dev);/*一个函数指针，当一个驱动程序所驱动的一
```




```
个设备被卸掉的时候被调用*/
        int (*save_state) (struct pci_dev *dev, u32 state);     /*在挂起前存储设备状态*/
        int (*suspend) (struct pci_dev *dev, u32 state);        /*把设备置于挂起状态*/
        int (*resume) (struct pci_dev *dev);                    /*唤醒设备*/
        int (*enable_wake) (struct pci_dev *dev, u32 state, int enable);   /*产生唤醒事件*/
};                                                              /*以上函数指针并不都是必需的*/
```

ID 列表是一个以全零入口结尾的 struct pci_device_id 类型的数组。

```
struct pci_device_id {
        unsigned int vendor, device;            /*厂商和设备 ID 号或者 PCI_ANY_ID */
        unsigned int subvendor, subdevice;      /*子系统 ID 号或者 PCI_ANY_ID */
        unsigned int class, class_mask;         /*设备类别号及设备类别掩码号，掩码号
告诉类别号的哪些位必须符合*/
        unsigned long driver_data;              /*驱动程序参数*/
};
```

例如，如果要为 sis 系列网卡编写驱动程序。

首先，定义 pci_driver 数据结构的变量 sis900_pci_driver，并为它初始化以注册设备。pci_driver 数据结构：

```
static struct pci_driver sis900_pci_driver = {
        name:           SIS900_MODULE_NAME,
        id_table:       sis900_pci_tbl,
        probe:          sis900_probe,
        remove:         sis900_remove,
};
```

注册设备：

```
static int __init sis900_init_module(void)
{
        printk(KERN_INFO "%s", version);
/*在下面的 pci_module_init()中会调用 pci_register_driver()注册设备*/
        return pci_module_init(&sis900_pci_driver);
}
```

如果退出驱动程序，要清除设备模块：

```
static void __exit sis900_cleanup_module(void)
```

```
{
        pci_unregister_driver(&sis900_pci_driver);
}
```

pci_module_init 是用来向 PCI 子系统注册一个 PCI 驱动程序，根据 id_table 中所提供的资料，PCI 子系统会在发现符合驱动程序要求的装置时使用它。那么 PCI 子系统如何做到这件事呢？接下来，通过查看 id_table 的内容就很清楚了。

```
static struct pci_device_id sis900_pci_tbl [] __devinitdata = {
 {PCI_VENDOR_ID_SI, PCI_DEVICE_ID_SI_900,
  PCI_ANY_ID, PCI_ANY_ID, 0, 0, SIS_900},
 {PCI_VENDOR_ID_SI, PCI_DEVICE_ID_SI_7016,
   PCI_ANY_ID, PCI_ANY_ID, 0, 0, SIS_7016},
     {0,}
};
MODULE_DEVICE_TABLE (pci, sis900_pci_tbl);
```

通过定义以上的数组，就表明这个驱动程序支持 SIS 公司出品的 sis900 和 sis7016 系列的所有硬件。

驱动程序退出的时候调用函数 pci_unregister_driver()，PCI 层会自动调用在 pci_driver 结构中注册的 remove 函数。

另外应把相应的函数设定为初始化或者清除类型。相应的宏在头文件 linux/linux-2.4.x include/linux/init.h 中定义：

- _init：初始化代码。驱动程序初始化完成后从内存中清除。
- _exit：退出代码。如果此驱动不是以模块的方式加载，则代码被忽略。
- _devinit：设备初始化代码。如果内核没有被配置为 CONFIG_HOTPLUG，即不支持热插拔，则与_init 相同，否则就是一个普通的函数。
- _devexit：与_exit 一样。

下面以 NE2000 的网卡驱动程序为例子，说明如何编写新型架构的网卡驱动程序。

12.3.2 嵌入式 Linux 下 PCI 驱动实例——NE2000 网卡驱动程序

前面已经对嵌入式 Linux 下 PCI 驱动的两种架构进行了说明，为了使读者对新型架构下 PCI 设备驱动程序有进一步了解。接下来，通过在嵌入式 Linux 下实现对 NE2000 型号的 PCI 网卡驱动程序加以说明。

NE2000 网卡是一种应用相当广泛的 PCI 网卡，它的芯片资料可以在网上找到。另外，由于它是一个网络设备，不同于字符设备或者块设备，网络设备的驱动程序有自己的特点，

但由于本章的重点是编写 PCI 驱动程序，对于这些特点，在此也不多加说明，有兴趣的读者可以参阅由 Alessandro Rubini 著的《Linux Device Divers》(《Linux 设备驱动程序》)一书。

由于整个驱动程序的实现代码太长，在这里只能介绍与 PCI 驱动有关的部分，整个驱动程序会在附带光盘中/demo/chap12/12-1 中完整给出。

（1）驱动程序应该包含的头文件：

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/errno.h>
#include <linux/pci.h>
#include <linux/init.h>
#include <linux/ethtool.h>

#include <asm/system.h>
#include <asm/io.h>
#include <asm/irq.h>
#include <asm/uaccess.h>

#include <linux/netdevice.h>
#include <linux/etherdevice.h>
/*在这个驱动程序中使用了 8390 网卡芯片，很多操作都是由公用的 8390 接口函数来进
行的*/
#include "8390.h"
```

（2）定义设备名称和版本号：

```
#define DRV_NAME        "ne2k-pci"
#define DRV_VERSION     "1.02"
#define DRV_RELDATE     "10/19/2000"
```

（3）定义此设备驱动程序支持的硬件：

```
static struct pci_device_id ne2k_pci_tbl[] __devinitdata = {
        { 0x10ec, 0x8029, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_RealTek_RTL_8029 },
        { 0x1050, 0x0940, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Winbond_89C940 },
        { 0x11f6, 0x1401, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Compex_RL2000 },
        { 0x8e2e, 0x3000, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_KTI_ET32P2 },
        { 0x4a14, 0x5000, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_NetVin_NV5000SC },
```

```
        { 0x1106, 0x0926, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Via_86C926 },
        { 0x10bd, 0x0e34, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_SureCom_NE34 },
        {0x1050,0x5a5a, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Winbond_W89C940F },
        { 0x12c3, 0x0058, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Holtek_HT80232 },
        { 0x12c3, 0x5598, PCI_ANY_ID, PCI_ANY_ID, 0, 0, CH_Holtek_HT80229 },
        { 0, }
};
MODULE_DEVICE_TABLE(pci, ne2k_pci_tbl);
```

（4）定义 pci_driver 结构的数据，注册探测函数和卸载函数入口：

```
static struct pci_driver ne2k_driver = {
        name:           DRV_NAME,
        probe:          ne2k_pci_init_one,   /*探测函数入口*/
        remove:         __devexit_p(ne2k_pci_remove_one),     /*卸载函数入口*/
        id_table:       ne2k_pci_tbl,
};
```

（5）注册模块初始化和模块清除函数：

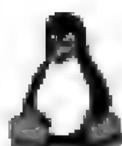
```
module_init(ne2k_pci_init);
module_exit(ne2k_pci_cleanup);
```

（6）注册设备：

```
static int __init ne2k_pci_init(void)
{
/* when a module, this is printed whether or not devices are found in probe */
#ifdef MODULE
        printk(version);
#endif
        return pci_module_init (&ne2k_driver);
}
```

（7）撤消设备，只有在模块化方式时退出设备驱动才调用：

```
static void __exit ne2k_pci_cleanup(void)
{
        pci_unregister_driver (&ne2k_driver);
}
```



（8）下面进入设备探测函数，这个函数会对前面定义的硬件设备进行探测，并会定义 struct net_device 结构的数据，同时对这个数据结构和 ei_status 结构进行填充。然后对满足条件的设备进行初始化，注册对它们的操作。

```
static int __devinit ne2k_pci_init_one (struct pci_dev *pdev,
                                const struct pci_device_id *ent)
{
        struct net_device dev;                  /*网卡设备描述符*/
        int i;
        unsigned char SA_prom[32];
        int start_page, stop_page;
        int irq, reg0, chip_idx = ent->driver_data;
        static unsigned int fnd_cnt;            /*发现的设备数*/
        long ioaddr;                            /*I/O 端口基地址*/
        int flags = pci_clone_list[chip_idx].flags;
/*如果编译进内核，只打印版本号*/
#ifndef MODULE
        static int printed_version;
        if (!printed_version++)
                printk(version);
#endif
/*设备号自动加 1*/
        fnd_cnt++;
/*通过配置空间的设置命令寄存器使能设备*/
        i = pci_enable_device (pdev);
        if (i)
                return i;
/*得到 I/O 基地址*/
        ioaddr = pci_resource_start (pdev, 0);
        irq = pdev->irq;        /*中断号*/
/*对 I/O 资源进行验证*/
        if (!ioaddr || ((pci_resource_flags (pdev, 0) & IORESOURCE_IO) == 0)) {
                printk (KERN_ERR PFX "no I/O resource at PCI BAR #0\n");
                return -ENODEV;
        }
        if (request_region (ioaddr, NE_IO_EXTENT, DRV_NAME) == NULL) {
                printk (KERN_ERR PFX "I/O resource 0x%x @ 0x%lx busy\n",
                        NE_IO_EXTENT, ioaddr);
                return -EBUSY;
        }
        reg0 = inb(ioaddr);
        if (reg0 == 0xFF)
```

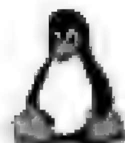
```
            goto err_out_free_res;

        /*验证设备有一块 8390 芯片*/
        . . .
/*为 dev 分配空间*/
        dev = alloc_etherdev(0);
        if (!dev) {
            printk (KERN_ERR PFX "cannot allocate ethernet device\n");
            goto err_out_free_res;
        }
        SET_MODULE_OWNER(dev);

        /* Reset 设备，并进行初始化*/
        . . .
        /*填充 dev 的值*/
        dev->irq = irq;
        dev->base_addr = ioaddr;
        pci_set_drvdata(pdev, dev);

        /* Allocate dev->priv and fill in 8390 specific dev fields. */
        if (ethdev_init(dev)) {
            printk (KERN_ERR "ne2kpci(%s): unable to get memory for dev->priv.\n",
                    pdev->slot_name);
            goto err_out_free_netdev;
        }
/*填充 ei_status 数据结构，注册设备操作函数*/
        ei_status.name = pci_clone_list[chip_idx].name;
        ei_status.tx_start_page = start_page;
        ei_status.stop_page = stop_page;
        ei_status.word16 = 1;
        ei_status.ne2k_flags = flags;
        if (fnd_cnt < MAX_UNITS) {
            if (full_duplex[fnd_cnt] > 0  ||  (options[fnd_cnt] & FORCE_FDX))
                ei_status.ne2k_flags |= FORCE_FDX;
        }

        ei_status.rx_start_page = start_page + TX_PAGES;
#ifdef PACKETBUF_MEMSIZE
        /* Allow the packet buffer size to be overridden by know-it-alls. */
        ei_status.stop_page = ei_status.tx_start_page + PACKETBUF_MEMSIZE;
#endif
```




```
        ei_status.reset_8390 = &ne2k_pci_reset_8390;
        ei_status.block_input = &ne2k_pci_block_input;
        ei_status.block_output = &ne2k_pci_block_output;
        ei_status.get_8390_hdr = &ne2k_pci_get_8390_hdr;
        ei_status.priv = (unsigned long) pdev;
        dev->open = &ne2k_pci_open;
        dev->stop = &ne2k_pci_close;
        dev->do_ioctl = &netdev_ioctl;
        NS8390_init(dev, 0);

        i = register_netdev(dev);
        if (i)
            goto err_out_free_8390;

        printk("%s: %s found at %#lx, IRQ %d, ",
                dev->name, pci_clone_list[chip_idx].name, ioaddr, dev->irq);
        for(i = 0; i < 6; i++) {
            printk("%2.2X%s", SA_prom[i], i == 5 ? ".\n": ":");
            dev->dev_addr[i] = SA_prom[i];
}
```

（9）下面是操作函数的实现，它们在上一步已经注册。包括打开操作、关闭操作、重启操作、得到硬件信息操作、输入输出操作以及 IOCTL 操作。主要牵涉到各寄存器的操作，具体实现在这里将不再加以说明，请参看 uClinux 源代码：

```
/*打开操作*/
static int ne2k_pci_open(struct net_device *dev){…}
/*关闭操作*/
static int ne2k_pci_close(struct net_device *dev){…}
/*重启操作*/
static void ne2k_pci_reset_8390(struct net_device *dev){…}
/*得到硬件信息操作*/
static void ne2k_pci_get_8390_hdr(struct net_device *dev, struct e8390_pkt_hdr *hdr,
                        int ring_page){…}
/*接收数据*/
static void ne2k_pci_block_input(struct net_device *dev, int count,
                        struct sk_buff *skb, int ring_offset){…}
/*发送数据*/
static void ne2k_pci_block_output(struct net_device *dev, const int count,
                const unsigned char *buf, const int start_page){…}
/*IOCTL 操作*/
static int netdev_ioctl(struct net_device *dev, struct ifreq *rq, int cmd){…}
```



12.4 小结

作为当前最流行的 PC 机系统总线，PCI 总线在嵌入式系统中也有广泛的应用，它解决了老式的 ISA 总线存在的诸多问题。

在本章第 1 节中介绍了 PCI 总线所有的相关特性。学习 PCI 总线，最重要的就是领悟掌握 PCI 总线自动配置的相关特性。

为了实现 PCI 自动配置，必须让设备告诉操作系统设备自身的信息，包括生产厂商、设备类型、希望有多大的存储器和 I/O 空间、映射到哪个地方、是否使用中断、设备工作的方式、工作状态等。这些都依赖于设备自带的寄存器空间——配置寄存器空间，所以在本章第 2 节中，对 PCI 配置空间进行了详细说明。另外要实现 PCI 规范，必须在操作系统中加入对它的支持，包括扫描设备、根据配置空间的内容为设备分配资源等。

编写 PCI 驱动程序，是嵌入式系统开发人员经常要做的工作。在嵌入式 Linux 下编写 PCI 驱动程序相当灵活，操作系统已经提供了很多的 API 供程序员使用。在本章第 3 节中介绍了编写 PCI 驱动程序的两种架构：一种是自己查找设备，注册设备的架构；另一种新型的架构采用 PCI 层来查找设置，而不用自己查找设备。在本节的最后通过编写一个 PCI 网卡的驱动程序介绍了编写 PCI 设备驱动程序的方法。

12.5 思考题

1．与 ISA 总线相比，PCI 总线有何特点？

2．PCI 主设备和从设备有何区别？

3．请参看 PCI2.1 规范，弄清它的引脚与时序图。

4．如果一个主设备奇偶校验出错，它会如何处理？

5．在 PCI 规范中，系统是如何知道一个设备是否存在的？

6．如果 PCI 设备想要在 32 位地址空间中申请 8MB 存储器地址空间，32KB I/O 地址空间，它的基地址寄存器该如何设置？

7．请画出嵌入式 Linux 初始化过程中对 PCI 设备的扫描和资源分配流程图。

第 13 章　嵌入式 Linux 网络编程

知识点：

- TCP/IP 协议简介
- 嵌入式 Linux 网络体系结构
- 嵌入式 Linux 中 socket 编程
- socket 编程高级特性
- socket 编程实例

本章导读：

本章对嵌入式 Linux 下网络编程进行介绍，讨论嵌入式设备通过 TCP/IP 协议与外部设备的通信问题。首先介绍 Linux 的网络体系结构，嵌入式 Linux 的网络体系结构与它是相似的，只是按照应用的需要进行了裁减而已。将重点介绍 TCP/IP 协议，因为这是目前应用最为广泛的网络通信协议。然后对嵌入式 Linux 环境下的 socket 编程进行说明，包括建立通信的过程、各 API 函数的功能，以及有连接和无连接通信的特点等。最后，作为应用总结，本章将举一个编写代理服务器的实例进行说明。



13.1 嵌入式 Linux 网络体系结构

Linux 是和网络密切相关的。从某种意义来说，Linux 是一个诞生于 Internet 和 WWW 的产品，它的开发者和用户用 Web 来交换信息、思想、程序代码。Linux 自身也常常被用来支持各种应用的网络需求。在目前"Every thing on line"的趋势下，任何成熟的嵌入式操作系统中都会集成或多或少的网络功能。

Linux 网络系统具有稳定、高效率、功能齐全和兼容范围广等特点。其设计简捷直观，并且支持多种网络协议，如 Ipv4、Ipv6、X.25、IPX、NETBIOS、DDP 等。下面将重点讨论 Linux 系统是如何支持 TCP/IP 协议的。

13.1.1 TCP/IP 网络简介

TCP/IP 协议最初只是用于支持计算机和 ARPANET 网络之间通信，但是随着时间的推移，它却被广泛应用于其他领域，使得其已经发展为目前事实上的网络标准协议。在 UNIX 系统中，首先带有网络功能的版本是 4.3 BSD。而 Linux 系统的网络功能就是以 UNIX 4.3 BSD 为模型发展起来的，所以它支持 BSD 套接口和全部的 TCP/IP 功能。这使得 UNIX 系统中的软件可以十分方便地移植到 Linux 系统中。

TCP/IP 参考模型是计算机网络的始祖，它首先提出了网络分层的概念。它一共分为 4 层：网络接口层、互联网层、传输层和应用层，其参考模型如图 13-1 所示。

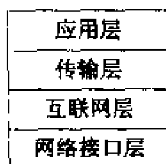


图 13-1 TCP/IP 参考模型

下面对 TCP/IP 参考模型中的各层加以说明。

1. 网络接口层

在 TCP/IP 参考模型中，它位于最底层，在 TCP/IP 协议中没有对这一层进行详细阐述。它主要实现将上层传下来的数据包封装成底层物理网络支持的数据格式，并转换成真正的电气信号而在物理网络上传输。

这一层其实对应于 7 层网络协议的物理层加上数据链路层，至于它的数据链路层，可以是以太网，也可以是 PPP、X.25 等网络。

2. 互联网层

在 TCP/IP 参考模型中，最重要的就是互联网层，它是整个 TCP/IP 网络系统的关键部



分。它的功能是使主机能够把分组发往任何网络，并使各分组独立地到达目标，也就是说，各分组所走的路径可以不一样，到达的顺序也不一定是发送的顺序，因此，如果需要按顺序发送和接收时，高层必须对分组进行排序。

互联网层定义了正式的分组格式和协议，即 IP 协议。在这一层最重要的工作是把 IP 分组发送到指定的地方去，由于 IP 分组可能要穿越多个连在一起的网络，并且网络连接状况很复杂，所以在这一层，分组路由和避免阻塞就是它所要考虑的主要问题。

为了表示 IP 分组的源和目的主机分别属于哪个网络，是网络中的哪台主机，在 IP 层使用 IP 地址和网络掩码来对其进行标识。在一个 TCP/IP 网络中，每台主机都分配有一个 32 位的 IP 地址，该地址可以惟一地标识主机。IP 地址通常是采用以“·”隔开的 4 个十进制数来表示，如 IP 地址若为 0x81124C15（16 进制）通常写成 129. 8.76.21。

IP 地址由两部分组成：网络（network）地址和主机（host）地址。网络地址由 IP 地址的高位组成，主机地址由低位组成，这两部分的大小取决于网络的类型。如一个 B 类地址（IP 地址的第一个字节大小在 128~191 之间），其 IP 地址的前两个字节是网络地址，后两个字节表示主机地址，这样一个 B 类地址可支持 65536 个网络，同时每个网络中可容纳 65536 台主机。

3. 传输层

由于在网络上的一台主机可能同时与外部有多个连接，这些连接又可能处于不同的状态（就绪、停止和传输中等状态），主机必须对这些连接进行管理，知道每个连接所处的状态，以及每个连接属于主机中正在运行的哪个进程，以便把接收到的数据包传送给相应的进程。要完成这些工作，就必须在 IP 层上面增加一层，这就是传输层。传输层的主要功能是使源端和目的端上的对等实体可以进行对话。

传输层是管理连接中端到端通信的协议层，为了表示各个端口，这一层使用了“端口号”的概念，用不同的端口号表示不同的端口。端口号的大小为 0~65535，有许多端口号已经固定为一些特定的上层应用使用，如 WWW 应用使用的是 80 端口。在传输层的连接中，不但要指定对端实体的 IP 地址，还要指定对端实体的端口号。

在这一层定义了两个端到端的协议：一个是传输控制协议 TCP，它是一个面向连接的协议，允许从一台机器发出的字节流无差错地到达另一台。它将输入的字节流分成报文段并传送给互联网层。TCP 还要处理流量控制，以控制因为发送方发送数据速度过快而产生数据包丢失；另一个协议是用户数据报协议 UDP，它是一个不可靠、无连接的协议，用于不需要 TCP 排序和流量控制的应用中，尽管可能不可靠，但是它具有连接过程简单、传送速度更快的特点，在网络状况较好时，可以使用它来传递数据。

TCP 建立在 IP 之上（这正是 TCP/IP 的由来），定义了网络上程序到程序的数据传输格式和规则，提供了 IP 数据包的传输确认、丢失数据包的重新请求、将收到的数据包按照它们的发送次序重新装配的机制。TCP 协议是面向连接的协议，类似于打电话，在开始传输数据之前，必须先建立明确的连接。

UDP 也建立在 IP 之上，但它是一种无连接协议，两台计算机之间的传输类似于传递邮件：消息从一台计算机发送到另一台计算机，两者之间没有明确的连接。UDP 中的 Datagram



是一种自带寻址信息，独立地从数据源走到终点的数据包。

4. 应用层

应用层协议建立在网络层协议之上，专门用于为用户提供应用服务，一般是可见的。如利用 FTP（文件传输协议）传输一个文件请求一个和目标计算机的连接，在传输文件的过程中，用户和远程计算机交换的一部分是能看到的。常见的应用层协议有：HTTP、FTP、Telnet、SMTP 和 Gopher 等。

● HTTP（Hyper Text Transport Protocol：超文本传输协议）

HTTP（Hyper Text Transport Protocol：超文本传输协议）是一个通用的、面向对象的协议，在 Internet 上进行信息传输时广泛使用。通过扩展请求命令，它可以用来实现许多任务。HTTP 允许系统相对独立于数据的传输，包括对该服务器上指定文件的浏览、下载、运行等。HTTP 不断发展，支持的媒体越来越多，使得可以方便地访问 Internet 上的各种资源。

● FTP（File Transfer Protocol：文件传输协议）

通过 FTP 可以实现从一个系统向另一个系统传输文件。通过 FTP 用户不仅可以方便地连接到远程服务器上，查看远程服务器上的文件内容，还可以把所需要的内容复制到自己所使用的计算机上；如果服务器允许用户对该服务器上的文件进行管理，该用户不仅可以把自己计算机上的文件传送到文件服务器上，让其他用户共享，还能自由地对服务器上的文件进行编辑操作，例如对文件进行删除、移动、复制、更名等。

● Telnet（远程登录协议）

Telnet 提供了一个相当通用的、双向的、面向 8 位字节的通信机制，使用基于文本界面的命令连接并控制远程计算机。Telnet 允许用户把自己的计算机当作远程主机上的一个终端，通过该协议用户可以登录到远程服务器上。用户通过 Telnet 登录到远程计算机后，便可以通过自己本地的计算机来控制和管理远程服务器上的文件及其他资源。

● SMTP（简单邮件传输协议）

这个协议可以实现可靠和高效的邮件传输。当用户给 SMTP 服务器发送请求时，一个双向的连接便建立起来，客户发一个 MAIL 指令，指示它想给 Internet 上的某处的一个收件人发个信。如果 SMTP 服务器允许这个操作，就会有一个肯定的确认发回客户机，随后会话开始。客户应当告知收件人的名称和 IP 地址，以及要发送的消息。

● Gopher（一种信息查询系统协议）

Gopher 相当于一个分布式的文件获取系统。文档放在许多服务器上，Gopher 客户软件给客户提供一个层次结构的项和目录，看上去像一个文件系统。Gopher 服务功能相当强大，能提供文本、声音和其他媒体。

在 TCP/IP 模型中，上层功能的实现要依赖下层提供的服务，例如在底层是以太网的 TCP/IP 网络体系体系中，IP 分组的传送要封装在以太网帧中，而 TCP/UDP 包的传送又是封装在 IP 分组的 data 中实现的。它们的封装关系如图 13-2 所示。

从图 13-2 中可以看出，上层协议的数据、地址、校验和等都是作为一个整体封装在下层协议数据包的 data 段，作为下层要传递的数据部分，而对等实体之间的通信只能看到本层协议的地址、校验和等信息，它不会试着辨认数据部分，只会把数据往上层送，它的上



层也是这样处理。这样依次剥离各层的附加信息，最终送到接收端应用层的就是发送端应用层发送的数据。

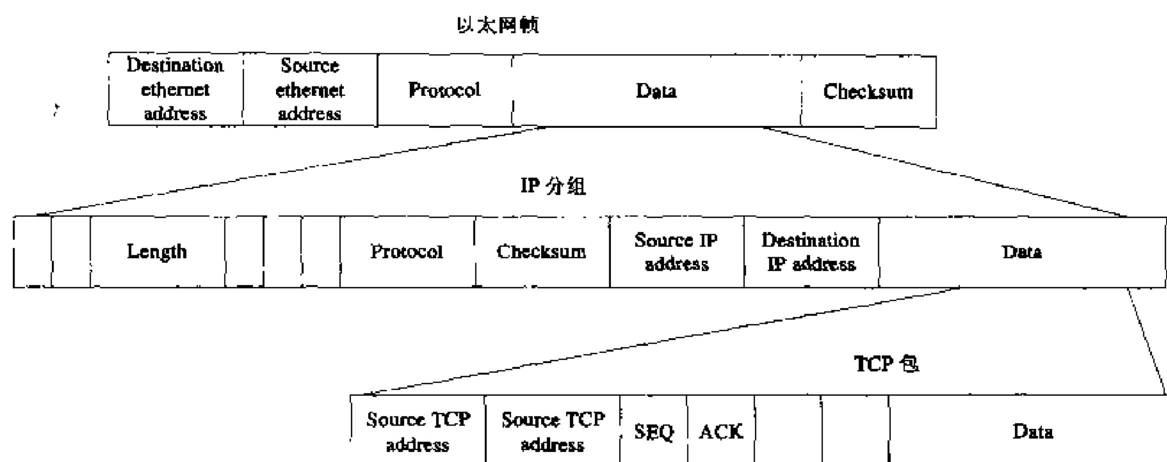


图 13-2 TCP/IP 协议各层的层次关系

但是各层的对等实体在通信时，除了要传递数据外，还必须告诉用户要传递给谁，这就需要加上目的地址。另外要告诉对端数据是谁发过来的，这就需要加上源地址。这个地址对于以太网帧来说就是 MAC 地址，对所有以太网设备来说它都是全球惟一的，共 48 位。对于 IP 分组来说就是 IP 地址，它是 32 位地址，在前面已经介绍过了。对于传输层的 TCP/UDP 包来说，标志地址的是 16 位的端口号。另外在各层的传输结构中，还要告诉对方本次传递的数据包的大小。为了进行差错控制，还要加上校验和字段。

由于在底层的局域网结构有以太网和 802.3 网络之分，所以要在数据帧中加以说明，这样在数据帧中增加了 protocol 字段加以标识。同理，由于 IP 分组所传递的上层传输层数据既可以是 TCP 数据包，又可以是 UDP 数据包，这样在 IP 分组中也增加了一个字段 protocol 来加以标识。

另外就是各层自己的一些用于控制和记录状态的字段。例如，在 IP 分组中有 SERVICE TYPE、FLAG 等字段，在 TCP 包中有标明发送和接收窗口大小、当前序号、确认序号、选项以及首部长度等字段，这些都是为了实现控制与记录状态而添加的字段。

13.1.2 嵌入式 Linux 中 TCP/IP 网络结构

Linux 用一系列相互连接层的软件来实现 Internet 协议地址族，它对 IP 协议族的实现机制如图 13-3 所示。其中 BSD 套接字（BSD socket）由专门处理 BSD socket 的通用套接字管理软件来处理，它由 INET socket 层来支持，这一层为基于 IP 的协议 TCP 和 UDP 提供端到端传输管理。

前面曾经说明，UDP（用户数据报协议）是一个无连接协议而 TCP（传输控制协议）是个可靠的端对端协议。传输 UDP 数据包时，Linux 不知道也不关心它们是否已经安全到

达目的地，而传输 TCP 数据包时，则被 TCP 连接两端编号以保证传输的数据被正确接收。

IP 层包含了实现 Internet 协议的代码。这些代码为要传输的数据加上 IP 头，并把传入的 IP 包送给 TCP 或 UDP。

在 IP 层以下，是支持所有 Linux 网络应用的网络设备层，如 PPP 和以太网。网络设备不总是物理设备，也包括如 loopback 一样的纯软件设备。注意，网络设备与字符设备或者块设备等标准设备的不同之处在于，标准的 Linux 设备用 mknod 命令加以建立，并且可以立即查看；而网络设备要用底层软件发现并加以初始化，它只能在系统初始化时才能被发现。例如，在建立一个有相应的以太网设备驱动在内的内核后，就可以看到/dev/eth0。ARP 协议位于 IP 层与支持 ARP 的底层协议之间，它负责把 IP 地址解释成底层网络设备可以辨认的地址，例如将 IP 地址解释成 MAC 地址。

Linux 网络部分沿用了传统的网络层次结构，网络数据从用户进程传输到网络设备总共需要经历 4 个层次，如图 13-4 所示。数据传输过程只能依照自上而下的顺序进行，不能跨越其中的某个或者某些层次，这就使得网络传输只能有惟一的一条路径，从而提高了整个网络的可靠性。

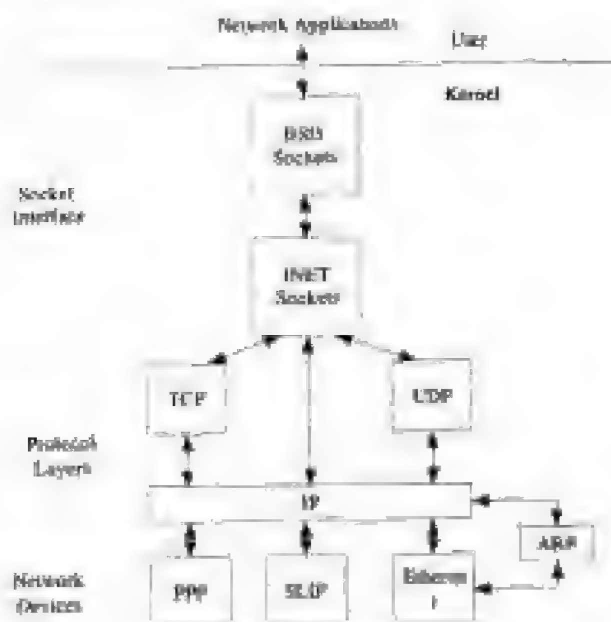


图 13-3　Linux 网络层次结构

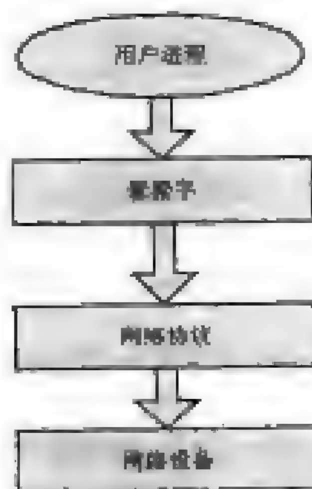


图 13-4　Linux 网络层次模型

13.2　嵌入式 Linux 环境下的 socket 编程

socket 在所有网络操作系统和网络应用程序中都是必不可少的，它是网络通信中应用进程和网络协议之间的接口。在 Linux 操作系统中，socket 属于文件系统的一部分，网络通信可以被看作是对文件的读取。这就使得用户对网络的控制像对文件的控制一样方便。下面就对嵌入式 Linux 下的套接字编程进行说明。



13.2.1 套接字接口

网络的 socket 数据传输是一种特殊的 I/O，socket 也是一种文件描述符，它也有类似于对文件操作的函数调用，如完成打开、读写等操作。在 TCP/IP 协议地址族中，按照提供服务的层次关系，套接字有 BSD 和 INTEN 之分，下面就对两者分别加以介绍。

1. BSD 套接字接口

socket 接口是为方便开发人员进行 TCP/IP 程序开发，而为 TCP/IP 协议所开发的一组应用程序接口。由于它最早应用于伯克利大学的 BSD UNIX 中，所以人们又把它称为 BSD socket（简称 BSD），BSD 套接字是一个通用的接口，它不仅支持各种网络工作形式，而且还是一个进程间通信机制。一个套接字描述一个通信连接的一端，在一个通信连接中的两端通信程序应各自有一个套接字来描述它们自己那一端。套接字可以被看成是一个专门的管道，但又不完全是管道，套接字对它们能容纳的数据量没有限制。Linux 支持多种类型的套接字地址族，如表 13-1 所示。

表 13-1　BSD 套接字地址族

地　址　族	说　　明
UNIX	UNIX 域套接字
INET	使用 TCP/IP 协议的 Internet 地址族
AX25	业余无线电 X25
IPX	适用于 Novell IPX 地址族
APPLETALK	适用于 Appletalk DDP 地址族
X25	适用于 X25 协议

Linux 将套接字地址族抽象为统一的 BSD 套接字接口，这个接口是应用程序的开发接口，由各地址族专有的软件支持，例如支持 Internet 地址族的软件是 TCP/IP 协议栈。Linux BSD socket 支持不同的通信类型，但并非所有的地址族都支持所有的服务类型。Linux BSD 套接字支持下列套接字类型：

- Stream

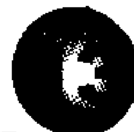
Stream 套接字提供可靠的双工顺序数据流，能保证传送过程中数据不丢失，不被弄混和复制。Internet 地址中的 TCP 协议支持流套接字。

- Datagram

Datagram 套接字提供双工数据传送，但与流套接字不同，它是不可靠的，不保证信息的到达。即使它们到达了，也不能保证其到达的顺序，甚至不能保证不被复制和弄混。这种套接字由 Internet 地址族中的 UDP 协议支持。

- Raw

利用这种类型的套接字可以直接处理下层协议（所以叫“Raw”）。例如，可以打开一



个 Raw 套接字到以太网设备，用于获取原始的 IP 数据包。

- Reliable Delivered Messages

Reliable Delivered Messages 套接字与数据报很像，但它能保证数据的到达。

- Sequenced Packets

Sequenced Packets 套接字与流套接字相似，但其数据包大小是固定的。

- Packet

这不是一个标准的 BSD 套接字类型，而是一个 Linux 特定的扩展，它允许在设备级上直接处理数据包。

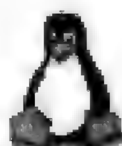
一个 BSD 操作的确切含义取决于底层的地址族。例如，设置一个 TCP/IP 连接就和设置一个业余无线电 X.25 连接有很大的不同。和 VFS（虚拟文件系统）一样，Linux 从 BSD 套接口协议层抽象出了套接口界面，此界面负责和各种不同的应用程序之间进行通信。内核初始化时，内核中的各个不同的地址族将会在 BSD 套接口界面中登记。稍后当应用程序创建和使用 BSD 套接口时，将会在 BSD 套接口和它支持的地址族之间建立一个连接。此连接是通过交叉关联的数据结构和地址族表建立的。例如，当一个应用程序创建一个新的套接口时，将产生一个可以被 BSD 套接口使用的与特定的地址族有关的套接口创建子过程。

设置系统内核时，一系列的地址族和协议将会保存在协议向量中。每一个协议都由它的名字代表，如 INET 和其初始化例程的地址。当系统启动并初始化套接口界面时，将会调用每一个协议的初始化例程。对于套接口地址族来说，这意味着注册了一系列有关协议的操作，这是一系列的子程序，每一个都要执行一个和特定的地址族有关的操作。

2. INET Socket 层

INET 套接口层包括支持 TCP/IP 协议的 Internet 地址族。正如上面提到的，这些协议是分层的，每一个协议都使用另一个协议的服务。Linux 系统中的 TCP/IP 代码和数据结构也反映了这种分层的思想。它和 BSD 套接口层的接口是通过一系列与 Internet 地址族有关的套接口操作来实现的，而这些套接口操作是在网络初始化的过程中由 INET 套接口层在 BSD 套接口层中注册的。这些操作和其他地址族的操作一样保存在 pops 向量中。BSD 套接口层通过 INET 的 proto_ops 数据结构来调用与 INET 层有关的套接口子程序，以实现有关 INET 层的服务。例如，一个地址族为 INET 的 BSD socket 建立请求，将用到下层的 INET socket 的建立函数。

BSD 套接口层将会把套接口数据结构传递给 INET 层。INET 套接口层在它自己的数据结构 sock 中而不是在 BSD 套接口的数据结构中插入有关 TCP/IP 的信息，但 sock 数据结构是和 BSD 套接口的数据结构有关的，通过图 13-5 可以看出这种相关关系。它使用 BSD 套接口中的数据指针来连接 sock 数据结构和 BSD 套接口数据结构，这意味着以后通过 INET 套接口调用可以十分方便地得到 sock 数据结构。数据结构 sock 中的协议操作指针也会在创建时设置好，并且此指针是和所需要的协议有关的。如果需要的是 TCP 协议，那么数据结构 sock 中的协议操作指针将会指向一系列的 TCP 协议操作。



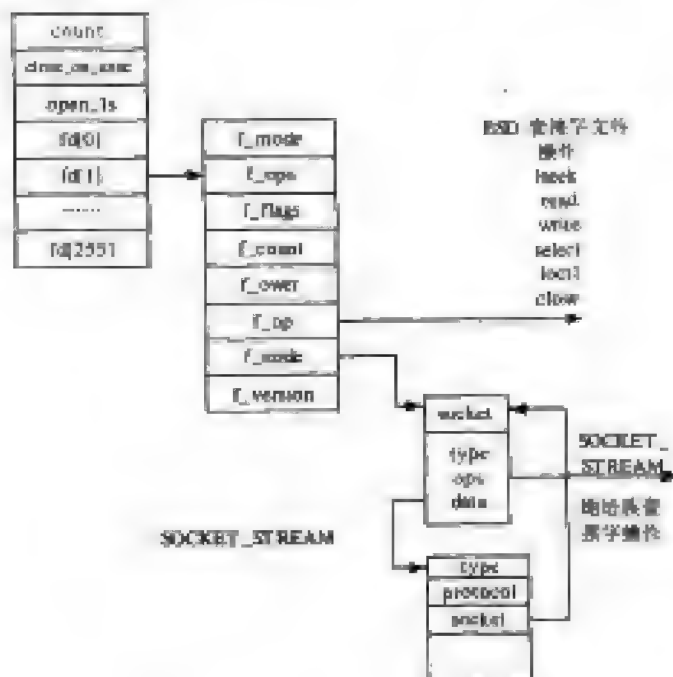


图 13-5　Linux BSD Socket 数据结构

socket 数据结构在 uClinux 源代码部分 linux/include/linux/net.h 中的定义如下所示：

```
struct socket
{
        socket_state            state;
/*定义为 SOCK_STREAM、SOCK_DGRAM 和 SOCK_RAM*/
        unsigned long           flags;
        struct proto_ops        *ops;
        struct inode            *inode;
        struct fasync_struct    *fasync_list;   /*Asynchronous wake up list*/
        struct file             *file;          /*File back pointer for gc*/
        struct sock             *sk;
        wait_queue_head_t       wait;
        short                   type;
        unsigned char           passcred;
};
```

而 sock 数据结构在 uClinux 源代码部分/include/net/sock.h 中定义，具体定义请参看 Linux 源代码中的相关定义。

socket 数据结构是面向进程和系统调用的数据结构，而 sock 结构是面向底层驱动程序的，两者存在链接关系，BSD socket 数据结构中的 data 指针将二者链接了起来，这使得以

后的 socket 操作能够比较方便地访问 sock 结构中的数据。

13.2.2 socket 编程基础

1. 客户机/服务器模式

在网络上大部分的通信都是在客户机/服务器模式下进行的，利用 socket 实现通信也不例外。在客户机/服务器模式中，将请求服务的一方称为客户端，而将提供服务的一方称为服务器端。例如 Telnet，当使用 Telnet 连接到远程主机的端口时，主机上的一个叫做 telnetd 的程序就开始运行。它将处理所有进入的 Telnet 连接，为连入的客户设置登录提示符等。

在 Internet 的应用中大多数应用都是比较单一的客户机/服务器模式，如 WWW 使用 HTTP 协议，由提供服务的任务侦听 80 端口，该任务就是服务的提供者，而浏览器就是客户端。也有一些应用程序，本身请求服务的同时也提供相应的服务，如果把功能拆分开来，这种程序也是客户机/服务器模式。例如，传送电子邮件的 SMTP 协议，接收/发送邮件的 Sendmail 程序，如果从接收邮件的功能来看，该程序就是一个邮件服务器，该程序侦听 25 端口，客户端可以通过客户端程序或者使用一些特定的命令将邮件发送到服务器端，这时 Sendmail 就是一个服务器端程序。但是当 Sendmail 向其他 SMTP 服务器转发邮件时，又可以将它看作为一个客户端程序。

在将网络编程模式定为客户机/服务器模式后，网络通信的程序就可以分为客户端程序和服务器程序，接下来将分别对这两方面加以说明。

2. socket 编程相关数据结构

下面将讨论使用套接口编写程序可能要用到的数据结构：

- 套接口描述符

一个套接口描述符只是一个整型的数值，它的定义为 int sockfd。

- struct sockaddr

这个数据结构中保存着套接口的地址信息，它的定义如下：

```
struct sockaddr {
unsigned short sa_family;          /*地址族：定义为 AF_xxx */
char sa_data[14];                  /*14 字节的协议地址*/
};
```

sa_family 中可以是其他的很多值，但一般把它赋值为“AF_INET”。
sa_data 则包括一个目的地址和一个端口地址。

- sockaddr _ in

sockaddr _ in 数据结构的定义为：




```
struct sockaddr_in {
short int sin_family;              /*地址族*/
unsigned short int sin_port;       /*端口号*/
struct in_addr sin_addr;           /* IP 地址*/
unsigned char sin_zero[8];         /*填充 0 以保持与 struct sockaddr 同样大小*/
};
```

这个数据结构使得使用其中的各个元素更为方便。要注意的是，sin_zero 应该使用 bzero() 或者 memset()以把它设置为全 0。另外，一个指向 sockaddr _in 数据结构的指针可以强行转换为一个指向数据结构 sockaddr 的指针，反之亦然。

3. 网络字节顺序

一个网络可能由不同体系结构的 CPU 组成，这些不同体系结构的 CPU 使用的字节顺序不同：有的 CPU 使用 big_endian（大端，在存储器中高字节存储在后），有的 CPU 使用 little_endian（小端，在存储器中高字节存储在前）。因此为了在网络间能够进行数据交换，需要对这些不同的字节顺序进行处理。

下面介绍几个字节顺序转换函数：

- htons()：表示"Host to Network Short"，把主机地址字节顺序转向网络字节顺序（对短型数据操作）。
- htonl()：表示"Host to Network Long"，把主机地址字节顺序转向网络字节顺序（对长型数据操作）。
- ntohs()：表示"Network to Host Short"，把网络字节顺序转向主机地址字节顺序（对短型数据操作）。
- ntohl()：表示"Network to Host Long"，把网络字节顺序转向主机地址字节顺序（对长型数据操作）。

13.2.3 socket 通信常用 API 函数

下面对 socket 通信中用到的函数进行详细说明。

1. socket( )

使用系统调用 socket( )来获得文件描述符，该调用的声明格式如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

其中，各个参数的含义如下：

- domain：说明网络程序所在的主机采用的通信协议（AF_UNIX 和 AF_INET 等）。AF_UNIX 只能用于单一的 UNIX 系统进程间通信，而 AF_INET 是针对 Internet 的，因而可以允许在远程主机之间通信，由于本书只讲述 TCP/IP 协议，所以只用到 AF_INET。
- Type：网络程序所采用的通信协议（SOCK_STREAM、SOCK_DGRAM 等）。其中 SOCK_STREAM 表明用的是 TCP 协议，这样会提供按顺序的、可靠的、双向的、面向连接的比特流；SOCK_DGRAM 表明用的是 UDP 协议，这样只会提供无序的、不可靠的、无连接的通信。
- Protocol：由于指定了 type，所以这里一般只要用 0 来代替即可。

socket()得到套接口描述符，为网络通信做基本的准备工作，调用成功时将返回文件描述符，失败时将返回-1，通过查看 errno 文件可以知道有关出错的详细信息。

2. bind()

一旦有了一个套接口以后，下一步工作就是把套接口绑定到本地计算机的某一个端口上，但如果只想使用 connect()则无此必要。

下面是系统调用 bind()的定义：

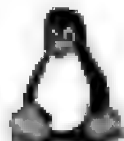
```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

其中，各个参数的含义如下：

- sockfd：是由 socket()调用返回的文件描述符。
- Addrlen：是 sockaddr 结构的长度。
- my_addr：是一个指向 sockaddr 的指针，在前面已经介绍过，不过由于系统的兼容性，一般不用这个头文件，而使用另外一个结构（struct sockaddr_in）来代替。

下面是一个使用 bind()的例子：

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#define MYPORT 3490
main ( )
{
int sockfd;
struct sockaddr_in my_addr;
sockfd = socket(AF_INET, SOCK_STREAM, 0); /* do some error checking! */
my _addr.sin_family = AF_INET; /* host byte order */
my_addr.sin_port = htons(MYPORT); /* short, network byte order */
```




```
my_addr.sin_addr.s_addr = inet_addr("132.241.5.10");
bzero (&(my_addr.sin_zero), 8); /* zero the rest of the struct */
/* don't forget your error checking for bind(): */
bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
...
```

如果出错，bind()也会返回-1。

3. connect()

该系统调用由客户端调用，connect ()的用法如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

其中，各个参数的含义如下：

- sockfd：socket 返回的文件描述符。
- serv_addr：储存了服务器端的连接信息，其中 sin_add 是服务端的地址。
- Addrlen：serv_addr 的长度，可以使用 sizeof(struct sockaddr)获得。

同样，如果出错，connect ()将会返回-1。

4. listen()

在服务器端，如果希望等待一个进入的连接请求，然后再处理这个连接请求，可以通过首先调用 listen()，然后再调用 accept()来实现。

系统调用 listen()的定义如下：

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

其中，各个参数的含义如下：

- sockfd：是调用过 bind()后的文件描述符。
- Backlog：设置请求排队的最大长度，当有多个客户端程序和服务器端相连时，使用这个表示可以接受的排队长度。由于进入的连接请求在使用 accept()应答之前要在进入队列中等待，这个值就是队列中最多可以拥有的请求的个数。大多数系统的默认设置为 20，可以设置为 5 或者 10。

当出错时，listen()将会返回-1 值。

5. accept()

当在远端的客户机试图使用 connect()连接服务器使用 listen()正在监听的端口时，此连

接将会在队列中等待，直到服务器使用 accept()处理它。调用 accept()之后，将会返回一个全新的套接口文件描述符来处理这个连接。这样，对于同一个连接来说，就有了两个文件描述符，原先的一个文件描述符还是监听指定的端口，而新的文件描述符可以用来进行数据传递。

accept()的用法如下：

```
#include <sys/socket.h>
int accept(int sockfd, void *addr, int *addrlen);
```

其中，各个参数的含义如下：

- sockfd：是 listen 后的文件描述符。
- addr、addrlen：这两个参数将会被客户端的程序填写，服务器端只要传递指针就可以了，它用来描述连接进来的客户端的信息。

accept()调用时，服务器端的程序会一直阻塞直到有一个客户程序发出了连接。accept()调用成功时，返回连接进来的客户机的文件描述符，之后服务器端就可以利用该描述符向客户机读写信息了，失败时返回-1。

6. send()和 recv()

这两个函数是在建立连接后用于完成发送与接收数据的系统调用，它们的用法为：

```
#include <sys/socket.h>
int send(int sockfd, const void *msg, int len, int flags);
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

其中各个参数的含义为：

- sockfd：希望进行数据传递的套接口的文件描述符，它可以是通过 socket()系统调用返回，也可以是通过 accept()系统调用得到。
- msg：指向希望发送的数据的指针。
- len：对于发送来说，它是希望发送的数据的字节长度。若是 recv()，则它表示接收缓冲的最大长度。
- flags：这个参数可以是 0 或者是表 13-2 中各项的组合。

表 13-2　flags 参数选项

宏 定 义	含 义
MSG_DONTROUTE	不查找路由表
MSG_OOB	接收或者发送带外数据
MSG_WAITALL	等待所有数据
MSG_PEEK	查看数据，并不从系统缓冲区移走数据



- MSG_DONTROUTE：是 send()函数使用的标志，这个标志告诉 IP 协议目的主机在本地网络上面，没有必要查找路由表。这个标志一般用在网络诊断和路由程序里面。
- MSG_OOB：表示可以接收和发送带外的数据，关于带外数据请参看 TCP/IP 相关书籍。
- MSG_PEEK：是 recv()函数的使用标志，表示只是从系统缓冲区中读取内容，而不改变系统缓冲区中的内容，这样下次读的时候，仍然是一样的内容。一般在有多个进程读写数据时可以使用这个标志。
- MSG_WAITALL：是 recv()函数的使用标志，表示等到所有的信息到达时才返回。使用这个标志的时候 recv()会一直阻塞，直到下面指定的条件满足，或者是发生了错误，才进行以下操作：
  - 当读到了指定的字节时，函数正常返回，返回值等于 len。
  - 当读到了文件的结尾时，函数正常返回，返回值小于 len。
  - 当操作发生错误时，返回-1，且设置错误为相应的错误号（errno）。

下面是一个简单使用 send()发送数据的例子：

```
char *msg = "lozen was here!";
int len, bytes_sent;
...
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
...
```

系统调用 send()返回实际发送的字节数，这可能比实际想要发送的字节数少。如果返回的字节数比要发送的字节数少，则在以后必须发送剩下的数据。当 send()出错时，将返回-1。系统调用 recv ()的使用方法和 send()类似。

7. sendto()和 recvfrom()

因为数据报套接口是无连接的，它并不连接到远程的主机上，所以在发送数据包之前，必须首先给出目的地址：

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);
```

除了最后两个参数以外，其他的参数含义与系统调用 send()时相同。其中，参数 to 是指向包含目的 IP 地址和端口号的数据结构 sockaddr 的指针。参数 tolen 可以设置为 sizeof(struct sockaddr)。

函数 sendto()用于返回实际发送的字节数，如果出错则返回-1。

函数 recvfrom()的使用方法和 recv()的使用方法十分近似：

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,struct sockaddr *from, int *fromlen);
```

其中，参数 from 是指向本地计算机中包含源 IP 地址和端口号的数据结构 sockaddr 的指针。参数 fromlen 设置为 sizeof(struct sockaddr)。如果对源地址信息不感兴趣，可以把它们设为 NULL。

系统调用 recvfrom()时返回接收到的字节数，如果出错则返回-1。

8. close()和 shutdown()

可以使用 close()调用关闭连接的套接口文件描述符，它的调用方式为：

```
close ( sockfd);
```

这样以后就不能再对此套接口进行任何的读写操作了。

使用系统调用 shutdown()，可有更多的控制权。它允许在某一个方向切断通信，或者切断双方的通信，它的调用方式为：

```
int shutdown(int sockfd, int how);
```

第 1 个参数是希望切断通信的套接口文件描述符。第 2 个参数 how 值如下：

- 0：不允许再接收数据。
- 1：不允许再发送数据。
- 2：发送和接收数据都不允许。

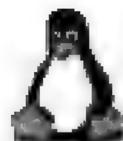
shutdown()调用如果成功返回 0，如果失败则返回-1。

13.2.4 数据流和数据报通信

前面已经介绍过有两种最常用的 Internet 套接口：数据流套接口和数据报套接口，经常采用"SOCK_STREAM"和"SOCK_DGRAM"代表上面两种套接口。数据流套接口是可靠的面向连接的通信数据流。如果在套接口中以"1，2"的顺序放入两个数据，它们在另一端也会以"1，2"的顺序到达，它们也可以被认为是无错误的传输。经常使用的 Telnet 应用程序就是使用数据流套接口的一个应用实例，另外，常用的 Web 浏览器也使用数据流套接口来读取网页。数据报套接口使用 UDP 来传送数据包，所以数据报的顺序是没有保障的。数据报是按一种应答的方式进行数据传输的。下面对这两种通信方式加以说明。

1. 数据流通信

整个面向连接的数据流通信的 socket 编程过程可以用图 13-6 来表示。



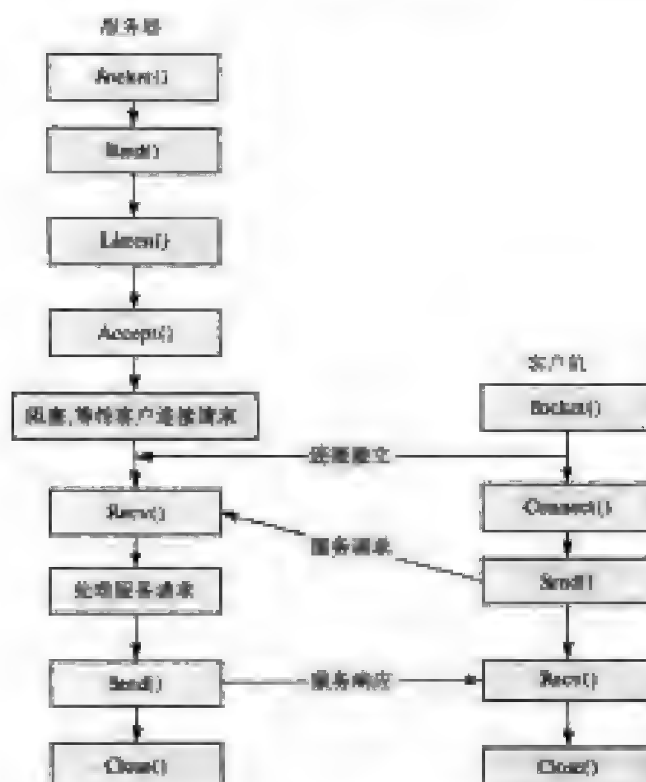


图 13-6　数据流通信过程

使用 socket 的数据流进行通信。首先，应该在服务器端使用 socket()建立一个通信的端点，再用 bind()命令把一个地址绑定到这个端点上。然后，服务器端使用 listen()倾听连接请求，当远程的客户机试图使用 connect()连接 listen()正在监听的端口时，连接将会在队列中等待，直到使用 accept()处理它，在 accept()处理了连接请求后，将会生成一个新的描述这个连接端口的套接字，利用这个套接字就可以发送和接收数据了。如果 listen()一直没有侦听到连接请求，那么服务器任务就会在 accept()处阻塞（在阻塞模式下），一直到有连接请求到来。

对于客户机任务来说，它也需要先用 socket()建立一个通信端口，但是它不必用 bind()把一个本地地址绑定到这个端口上，而是直接使用 connect()向指定的服务器发送连接请求，如果请求被接收，下一步就可以进行数据流通信了。

下面给出一个数据流通信的 Socket 编程实例（见光盘/demo/chap13/13-1）。在服务器端，它可以指定用于通信的端口，用法是#TCPServer xxx，其中 xxx 为端口号。当客户端连接进该服务器时，就发送给它“Hello! Socket communication world!”的语句。

```
/*******服务器端程序　TCPServer.c) ************/
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
```

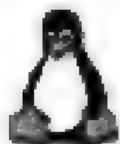
```
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#define WAITBUF 10
int main(int argc, char *argv[])
{
 int sockfd,new_fd;
 struct sockaddr_in server_addr;
 struct sockaddr_in client_addr;
 int sin_size,portnumber;
 char hello[]="Hello! Socket communication world!\n";

 if(argc!=2)
 {
  fprintf(stderr,"Usage:%s portnumber\a\n",argv[0]);
  exit(1);
 }
 /*端口号不对，退出*/
 if((portnumber=atoi(argv[1]))<0)
 {
  fprintf(stderr,"Usage:%s portnumber\a\n",argv[0]);
  exit(1);
 }

/*服务器端开始建立 socket 描述符*/
 if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)
 {
          fprintf(stderr,"Socket error:%s\n\a",strerror(errno));
          exit(1);
 }

/*服务器端填充 sockaddr 结构*/
 bzero(&server_addr,sizeof(struct sockaddr_in));
 server_addr.sin_family=AF_INET;
/*自动填充主机 IP*/
 server_addr.sin_addr.s_addr=htonl(INADDR_ANY);
 server_addr.sin_port=htons(portnumber);

/*捆绑 sockfd 描述符*/
 if(bind(sockfd,(struct sockaddr *)(&server_addr),sizeof(struct sockaddr))==-1)
```




```
	{
		fprintf(stderr,"Bind error:%s\n\a",strerror(errno));
		exit(1);
	}

/*监听 sockfd 描述符*/
	if(listen(sockfd, WAITBUF)==-1)
	{
		fprintf(stderr,"Listen error:%s\n\a",strerror(errno));
		exit(1);
	}

	while(1)
	{
		/*服务器阻塞，直到客户程序建立连接*/
		sin_size=sizeof(struct sockaddr_in);
		if((new_fd=accept(sockfd,(struct sockaddr *)(&client_addr),&sin_size))==-1)
		{
			fprintf(stderr,"Accept error:%s\n\a",strerror(errno));
			exit(1);
		}
	/*可以在这里加上自己的处理函数*/
		fprintf(stderr,"Server get connection from %s\n",
						inet_ntoa(client_addr.sin_addr));
		if(send(new_fd,hello,strlen(hello),0)==-1)
		{
		 fprintf(stderr,"Write Error:%s\n",strerror(errno));
		 exit(1);
		}
/*这个通信已经结束*/
		close(new_fd);
/*循环下一个*/
	}
	close(sockfd);
	exit(0);
}
```

下面是与之相连接的客户端程序，它的用法是#TCPClient ServerIP ServerPort，其中 ServerIP 是服务器的 IP 地址，ServerPort 是服务器开启的服务端口。

```
/******** 客户端程序  TCPClient.c ************/
#include <stdlib.h>
```

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#define RECVBUFSIZE 1024
int main(int argc, char *argv[])
{
  int sockfd;
  char buffer[RECVBUFSIZE];
  struct sockaddr_in server_addr;
  struct hostent *host;
  int portnumber,nbytes;

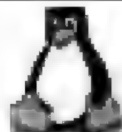
  if(argc!=3)
  {
   fprintf(stderr,"Usage:%s hostname portnumber\a\n",argv[0]);
   exit(1);
  }

  if((portnumber=atoi(argv[2]))<0)
  {
        fprintf(stderr,"Usage:%s hostname portnumber\a\n",argv[0]);
        exit(1);
  }

 /*客户端程序开始建立 sockfd 描述符*/
  if((sockfd=socket(AF_INET,SOCK_STREAM,0))==-1)
  {
        fprintf(stderr,"Socket Error:%s\a\n",strerror(errno));
        exit(1);
  }

 /*客户端程序填充服务端的资料*/
  bzero(&server_addr,sizeof(server_addr));
  server_addr.sin_family=AF_INET;
  server_addr.sin_port=htons(portnumber);
     server_addr.sin_addr.s_addr = inet_addr(argv[1]);

 /*客户端程序发起连接请求*/
```




```
if(connect(sockfd,(struct sockaddr *)(&server_addr),sizeof(struct sockaddr))==-1)
{
  fprintf(stderr,"Connect Error:%s\a\n",strerror(errno));
  exit(1);
}

/*连接成功*/
if((nbytes=recv(sockfd,buffer, RECVBUFSIZE),0)==-1)
{
          fprintf(stderr,"Read Error:%s\n",strerror(errno));
          exit(1);
 }
 buffer[nbytes]='\0';
 printf("I have received:%s\n",buffer);
/*结束通讯*/
 close(sockfd);
 exit(0);
}
```

2. 数据报通信

数据报通信的通信过程可以用图 13-7 来表示。

对于数据报通信的服务器端来说，它不必再在一个端口上侦听，以等待建立连接，而只需生成一个端口描述符，并且把这个端口描述符绑定到本地地址上就可以了。对于客户端也是一样。这样整个通信过程就简洁得多。需要说明的是，UDP 的客户端可以使用 connect()，但是这时使用 connect()并不真正产生连接，而只是填写对端套接字的有关信息。使用 connect()的好处是，随后的程序通信中不必每次指定地址，即可以使用 recv()、send()等进行通信。否则，就应该使用 recvfrom()、sendto()等实现函数通信，而每次都指定对端地址信息，如图 13-7 所示。

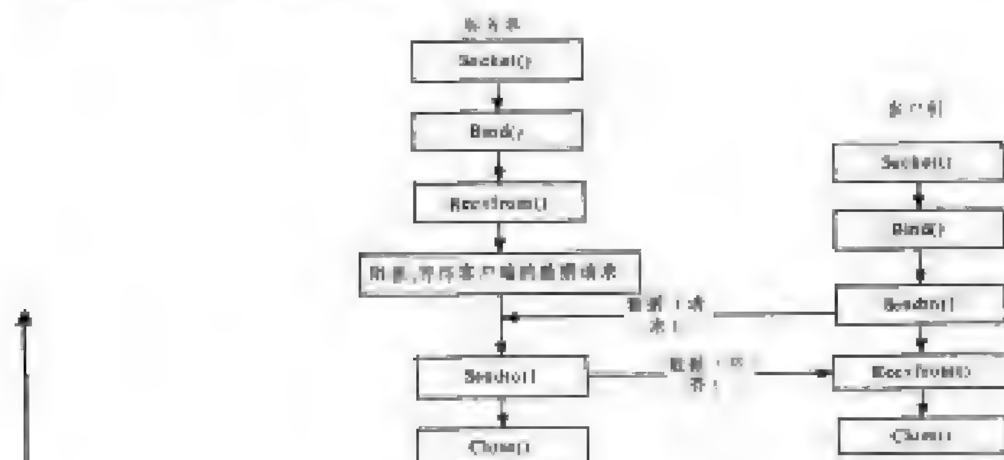


图 13-7　数据报通信过程

下面给出数据报通信中的服务器端和客户端程序实例（见光盘/demo/chap13/13-2）：

```
/*服务器端程序 UDPServer.c*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <errno.h>
#define SERVER_PORT        8888
#define MAX_MSG_SIZE       1024

void udps_respon(int sockfd)
{
        struct sockaddr_in addr;
        int      addrlen,n;
        char     msg[MAX_MSG_SIZE];

        while(1)
        {       /*等待数据请求*/
                n=recvfrom(sockfd,msg,MAX_MSG_SIZE,0,
                        (struct sockaddr*)&addr,&addrlen);
                msg[n]=0;
                /*显示服务器端已经收到了信息*/
                fprintf(stdout,"I have received %s",msg);
                 /*数据回送*/
                sendto(sockfd,msg,n,0,(struct sockaddr*)&addr,addrlen);
        }
}

int main(void)
{
        int sockfd;
        struct sockaddr_in         addr;

        sockfd=socket(AF_INET,SOCK_DGRAM,0);
        if(sockfd<0)
        {
                fprintf(stderr,"Socket Error:%s\n",strerror(errno));
                exit(1);
        }
        bzero(&addr,sizeof(struct sockaddr_in));
        addr.sin_family=AF_INET;
```




```
	addr.sin_addr.s_addr=htonl(INADDR_ANY);
	addr.sin_port=htons(SERVER_PORT);
	if(bind(sockfd,(struct sockaddr *)&addr,sizeof(struct sockaddr_in))<0)
	{
		fprintf(stderr,"Bind Error:%s\n",strerror(errno));
		exit(1);
	}
	udps_respon(sockfd);
	close(sockfd);
}

/*客户端程序 UDPClient.c；使用方法 UDPClient ServerIP ServerPort*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#define MAX_BUF_SIZE	1024

void udpc_requ(int sockfd,const struct sockaddr_in *addr,int len)
{
	char buffer[MAX_BUF_SIZE];
	int n;
	while(1)
	{		/*从键盘读入，写到服务器端*/
		fgets(buffer,MAX_BUF_SIZE,stdin);
		sendto(sockfd,buffer,strlen(buffer),0,addr,len);
		bzero(buffer,MAX_BUF_SIZE);
		/*从网络上读，写到屏幕上*/
		n=recvfrom(sockfd,buffer,MAX_BUF_SIZE,0,NULL,NULL);
		buffer[n]=0;
		fputs(buffer,stdout);
	}
}

int main(int argc,char **argv)
{
	int sockfd,port;
	struct sockaddr_in	addr;

	if(argc!=3)
```

```
    {
            fprintf(stderr,"Usage:%s server_ip server_port\n",argv[0]);
            exit(1);
    }

    if((port=atoi(argv[2]))<0)
    {
            fprintf(stderr,"Usage:%s server_ip server_port\n",argv[0]);
            exit(1);
    }

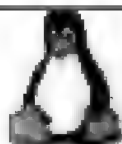
    sockfd=socket(AF_INET,SOCK_DGRAM,0);
    if(sockfd<0)
    {
            fprintf(stderr,"Socket   Error:%s\n",strerror(errno));
            exit(1);
    }
    /*填充服务器端的资料*/
    bzero(&addr,sizeof(struct sockaddr_in));
    addr.sin_family=AF_INET;
    addr.sin_port=htons(port);
    if(inet_aton(argv[1],&addr.sin_addr)<0)
    {
            fprintf(stderr,"Ip error:%s\n",strerror(errno));
            exit(1);
    }
    udpc_requ(sockfd,&addr,sizeof(struct sockaddr_in));
    close(sockfd);
}
```

13.2.5 socket 编程高级特性

在本章的 2.3 小节介绍了在 socket 通信中常用的 API 函数，利用这些 API 函数可以满足基本的 socket 通信需要，并且介绍了如何利用这些函数来进行面向连接的数据流和无连接的数据报通信。但是在 socket 应用编程中，还有许多高级应用特性，如获取服务器和客户机主机信息、进行阻塞处理、设置服务器工作模式、使用原始套接字等，本节将针对这些内容加以介绍。

1. 获取服务器和客户机主机信息

（1）IP 地址和域名的转换。



在网络上标识一台机器时既可采用 IP 或者也可采用域名，那么怎么实现两者的转换呢？可以使用如下函数：

```
struct hostent *gethostbyname(const char *hostname)
struct hostent *gethostbyaddr(const char *addr,int len,int type)
```

至于返回的数据结构 struct hostent，在</linux/linux-2.4.x/include/netdb.h>中有它的定义：

```
struct hostent{
        char *h_name;           /*主机的正式名称*/
        char *h_aliases;        /*主机的别名*/
        int   h_addrtype;       /*主机的地址类型 AF_INET*/
        int   h_length;         /*主机的地址长度，对于 IP4 是 4 个字节 32 位*/
        char **h_addr_list;     /*主机的 IP 地址列表*/
        }
 #define h_addr h_addr_list[0]  /*主机的第一个 IP 地址*/
```

gethostbyname()可以将机器名（如 linux.yessun.com）转换为一个 hostent 结构指针，在这个结构里面储存了域名的地址信息。

gethostbyaddr()可以将一个 32 位的 IP 地址（C0A80001）转换为结构指针，在这个结构里面储存了域名的地址信息。

这两个函数调用失败时返回 NULL，且设置 h_errno 错误变量。通过调用 h_strerror()可以得到详细的出错信息。

（2）字符串的 IP 地址和 32 位的 IP 地址转换。

在网络上使用的 IP 地址都是用数字加点的字符串表示（如"192.168.0.1"），而在 struct in_addr 结构中采用的却是 32 位的 IP 地址，如上面那个 IP 的字符串（"192.168.0.1"）应该表示为 C0A80001，为了在两者之间进行转换可以使用下面两个函数：

```
int inet_aton(const char *cp,struct in_addr *inp)
char *inet_ntoa(struct in_addr in)
```

其中，a 代表"ascii"，n 代表"network"。第 1 个函数表示将 a.b.c.d 的 IP 转换为 32 位的 IP 地址，存储在 inp 指针里面；第 2 个函数是指将 32 位 IP 地址转换为"a.b.c.d"的字符串格式。

（3）获取主机信息函数。

在网络程序里面有时需要知道另一端端口 IP 和主机名称信息，这时可以使用以下函数：

```
int getpeername(int sockfd,struct sockaddr *peeraddr, int *addrlen)
```

getpeername()函数将得到另一端的端口地址。sockfd 参数是连接的数据流套接口文件描述符。peeraddr 参数是指向包含另一端的信息的数据结构 sockaddr 的指针。addrlen 参数可以设置为 sizeof(struct sockaddr)。

如果出错，系统调用将返回-1。

```
int gethostname(char *name,int namelen);
```

系统调用 gethostname ()比系统调用 getpeername()还简单，它返回程序正在运行的计算机的名字，然后系统调用 gethostbyname ()就可以通过该计算机名得到 IP 地址。

2. 阻塞处理

在数据流通信中，当服务器运行到 accept()语句时，如果没有客户连接请求到来，那么会发生什么情况呢？这时服务器就会停止在 accept 语句上等待连接服务请求的到来。同样，当程序运行到接收数据语句时，如果这时接收缓冲区中没有数据可以读取，则程序也会停止在接收语句上。出现这种情况就称为阻塞（blocking），在前面对任务的介绍过程中曾讲过有关任务阻塞状态的相关知识。

当第一次创建一个套接口文件描述符时，系统内核默认将它设置为可以阻塞。如果不希望套接口阻塞，可以使用系统调用 fcntl()，如：

```
#include <unistd.h>
#include <fcntl.h>
...
sockfd = socket(AF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
...
```

如果设置为不阻塞，那么就得频繁地询问套接口以检查有无信息到来，这样会降低系统的效率。如果试图读取一个没有阻塞的套接口，同时它又没有数据，那么将返回-1。

由于询问套接口以检查有无信息的到来可能会占用太多的 CPU 时间，另一个可以使用的方法是 select ()。

select()用于同步 I/O 多路复用，这个系统调用十分有用。它允许用户把任务本身挂起来，同时使嵌入式 Linux 系统内核监听所要求的一组文件描述符的所有活动。只要确认在被监听的文件描述符上出现活动，select()调用将返回指示该文件描述符已经准备好的信息。这样就实现了使用 select()为任务探测出任何输入、输出变化，而不必由任务本身进行测试而浪费系统开销。系统通过调用 select()使得可以同时监视几个套接口，它可以指示哪一个套接口已经准备好了以供读取，哪一个套接口已经可以写入。

下面是 select()的用法：




```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int numfds, fd_set *readfds, fd_set *writefds,fd_set *exceptfds, struct timeval
*timeout);
```

其中的参数 readfds、writefds 和 exceptfds 分别是被 select()监视的读、写和异常处理的文件描述符。如果希望检查可以从标准输入或一些其他的套接口 sockfd 中读取数据，只需把文件描述符 0 或相应的 sockfd 的值赋给 readfds 即可。参数 numfds 应该设置为最高的文件描述符的值加 1。

当 select()返回时，readfds 将会被修改以便反映选择的那一个文件描述符已经准备好了以供读取，可以使用 FD_ISSET()加以测试。为了实现 fd_set 中对应的文件描述符的设置、复位和测试，它还有一组相应的宏：

- FD_ZERO(fd_set *set)：清除文件描述符集。
- FD_SET(int fd, fd_set *set)：把 fd 添加到文件描述符集中。
- FD_CLR(int fd, fd_set *set)：把 fd 从文件描述符集中移走。
- FD_ISSET(int fd, fd_set *set)：检测 fd 是否在文件描述符集中。

数据结构 timeval 在前面第 8 章中曾加以介绍，下面举一个使用 select()等待标准输入 2.5 秒的例子。

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#define STDIN 0 /* file descriptor for standard input */
main ( )
{
struct timeval tv;
fd_set readfds;
tv.tv_sec = 2;
tv.tv_usec = 500000;
FD_ZERO(&readfds) ;
FD_SET(STDIN, &readfds);/*任务在此阻塞*/
/* don't care about writefds and exceptfds: */
select(STDIN+1, &readfds, NULL, NULL, &tv);
if (FD_ISSET(STDIN, &readfds))
printf("A key was pressed!\n");
else
printf( " Timed out.\n");
}
```

使用 select()函数，可以有效地节省系统资源。

3. 服务器工作模式

不管是 TCP 还是 UDP 通信，在服务器端的程序结构设计上都可以使用两种基本模式：循环模式和并发模式。

在本章前面所举的实例都是采用循环模式，就是服务器端任务在总体结构上是一个循环，一次处理一个请求。这样，当有多个客户请求时，请求将被放入队列中，依次等待处理。但这样就产生一个处理时间问题，因为队列的长度是有限的，如果处理不过来将会导致等待队列溢出，这时就将不能处理新的连接请求。循环模式使用单任务结构。

并发模式的服务器进程可以同时处理多个请求，结构上一般采用父进程接收请求，然后调用 fork 产生子进程，由子进程处理请求，这一般是多任务结构。并发模式的优点是可以同时处理多个请求，客户端等待时间短。在并发模式设计中，也可以采用单任务结构，即用 select 调用来获取异步 I/O。单任务并发的优点是服务器端共享数据区，即对不同的客户不仅可以共享正文，而且可以共享数据，使得客户端之间交换数据易于实现。

4. 原始套接字

在前面已经学习过了网络程序的两种套接字：SOCK_STREAM 和 SOCK_DGRAM。在这里将介绍另外一种套接字——原始套接字（SOCK_RAW）。应用原始套接字，可以编写出由 TCP 和 UDP 套接字不能够实现的功能。

原始套接字允许对底层协议如 IP 或 ICMP 直接访问，可以直接填充 IP、TCP、UDP 或者 ICMP 的包头，发送用户自己定义的 IP 包或者 ICMP 包。它的功能强大，但使用较为不便，主要用于一些协议的开发，在网络安全的抓包中有重要的应用。

原始套接字的创建可以使用如下调用：

```
int sockfd(AF_INET,SOCK_RAW,protocol);
```

根据协议的类型不同，可以创建不同类型的原始套接字，比如：IPPROTO_ICMP、IPPROTO_TCP、IPPROTO_UDP 等。

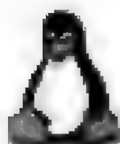
注意：原始套接字只能由拥有 root 权限的人创建。

13.3 网络编程实例——使用 socket 编写代理服务器

在 13.2 节中已经对在嵌入式 Linux 下如何应用 socket 进行网络通信进行了说明，本节使用一个实例来进一步说明。

13.3.1 功能说明

代理服务器是网络通信中常用的一种服务通信程序，它通过指定自身的一个端口执行



代理服务，以代理远端服务器的服务端口。例如，假如在主机名为“ProxyServer”的服务器 8000 端口上运行代理服务程序，代理远端主机“TelnetServer”上的 Telnet 服务，那么以后客户机如果要使用 TelnetServer 的 Telnet 服务，只需输入 Telnet ProxyServer 8000 即可。

在本例中，假如编译后生成了名为 Proxy 的可执行文件，那么命令及其参数的描述为：

./Proxy <proxy_port> <remote_host> <service_port>

其中，参数 proxy_port 表示指定的代理服务器端口。参数 remote_host 是指希望连接的远程主机的主机名（或 IP 地址）。这个主机名在网络上应该是惟一的，如果不能确定，可以在远程主机上使用 uname -n 命令查看。参数 service_port 是远程主机可提供的服务名，也可直接输入服务对应的端口号。这个命令的相应操作是将代理服务器的 proxy_port 端口绑定到 remote_host 的 service_port 端口。然后就可以通过代理服务器的 proxy_port 端口访问 remote_host 了。

例如一台远端 telnet 服务器，服务器名是 legends，IP 地址为 10.10.8.221，如果在本地计算机上执行：

#./proxy 8000 legends telnet

那么，就可以通过下面这条命令访问 legends 的 telnet 端口。

#telnet localhost 8000

前面的绑定代理端口操作也可以使用下面的命令：

#./proxy 8000 10.10.8.221 23

注意：23 是 telnet 服务的标准端口号，其他服务的对应端口号可以在/etc/services 中查看。

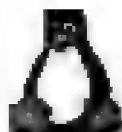
直接地说，代理服务器就是在远端服务器和客户机之间建立连接，服务器与客户机之间的通信通过代理服务器进行。

13.3.2 代码

以下是实现代理服务器的完整源代码（见光盘/demo/chap13/13-3）：

```
#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/file.h>
#include <sys/ioctl.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <netdb.h>
#define TCP_PROTO      "tcp"
```

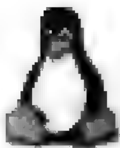
```
int proxy_port;                          /*全局变量，指定代理服务器的端口*/
struct sockaddr_in hostaddr;             /*全局变量，远端主机地址*/
extern int errno;
extern char *sys_myerrlist[];
void parse_args (int argc, char **argv);    /*参数转换函数*/
void daemonize (int servfd);                /*创建守护进程函数*/
void do_proxy (int usersockfd);             /*代理处理函数*/
void errorout (char msg);                   /*错误输出函数*/
/*********************************************************
主函数
*********************************************************/
main (argc,argv)
int argc;
char **argv;
{
        int clilen;
        int childpid;
        int sockfd, newsockfd;
        struct sockaddr_in servaddr, cliaddr;
/*把命令行参数转存到全局变量中*/
        parse_args(argc,argv);
        /*为侦听客户请求准备一个地址*/
        bzero((char *) &servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
        servaddr.sin_port = proxy_port;
        /*得到一个端口的文件描述符*/
        if ((sockfd = socket(AF_INET,SOCK_STREAM,0)) < 0) {
            fputs("failed to create server socket\r\n",stderr);
            exit(1);
        }
        /*绑定到前面的地址上*/
        if      (bind(sockfd,(struct sockaddr_in *) &servaddr,sizeof(servaddr)) < 0) {
            fputs("faild to bind server socket to specified port\r\n",stderr);
            exit(1);
         }
        /*准备接收*/
        listen(sockfd,5);
    /*把自身变为守护进程*/
    daemonize(sockfd);
    /*进入一个循环，并发处理连接请求*/
    while (1) {
```




```
        /*接受连接请求*/
        clilen = sizeof(cliaddr);
        newsockfd = accept(sockfd, (struct sockaddr_in *) &cliaddr, &clilen);
        if (newsockfd < 0 && errno == EINTR)
            continue;
          else if (newsockfd < 0)
            /*出错，kill 服务器*/
        errorout("failed to accept connection");
        /*产生一个子进程，进行连接处理*/
        if ((childpid = fork()) == 0) {
            close(sockfd);
            do_proxy(newsockfd);          /*真正的处理过程*/
            exit(0);
          }
        /*如果产生子进程失败，连接将被丢掉*/
          close(newsockfd);
          }
     }
/***********************************************************************
进行参数转换，把从命令行得到的参数值赋给全局变量
***********************************************************************/
void parse_args (argc,argv)
int argc;
char **argv;
{
    int i;
    struct hostent *hostp;
    struct servent *servp;
    unsigned long inaddr;
    struct {
        char proxy_port [16];
        char isolated_host [64];
        char service_name [32];
    } pargs;
/*输入不合规范*/
    if (argc < 4) {
          printf("usage: %s <proxy-port> <host> <service-name|port-number>\r\n", argv[0]);
          exit(1);
    }
/*将输入参数先放到自定义的数据结构中*/
    strcpy(pargs.proxy_port,argv[1]);
    strcpy(pargs.isolated_host,argv[2]);
```

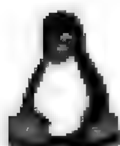


```
    strcpy(pargs.service_name,argv[3]);
/*检查端口号是否是数字，再赋给 proxy_port*/
    for (i = 0; i < strlen(pargs.proxy_port); i++)
        if (!isdigit(*(pargs.proxy_port + i)))
            break;
    if (i == strlen(pargs.proxy_port))
        proxy_port = htons(atoi(pargs.proxy_port));
    else {
        printf("%s: invalid proxy port\r\n",pargs.proxy_port);
        exit(0);
    }
/*把远端服务器地址赋给 hostaddr*/
    bzero(&hostaddr,sizeof(hostaddr));
    hostaddr.sin_family = AF_INET;
/*不管是主机名还是 IP 地址，都把它转换为 hostaddr 的地址*/
    if ((inaddr = inet_addr(pargs.isolated_host)) != INADDR_NONE)
        bcopy(&inaddr,&hostaddr.sin_addr,sizeof(inaddr));
    else if ((hostp = gethostbyname(pargs.isolated_host)) != NULL)
        bcopy(hostp->h_addr,&hostaddr.sin_addr,hostp->h_length);
    else {
        printf("%s: unknown host\r\n",pargs.isolated_host);
        exit(1);
    }
/*不管是用数字表示的端口还是用服务表示的端口，都把它转换后赋给 hostaddr.sin_port*/
    if ((servp = getservbyname(pargs.service_name,TCP_PROTO)) != NULL)
        hostaddr.sin_port = servp->s_port;
    else if (atoi(pargs.service_name) > 0)
        hostaddr.sin_port = htons(atoi(pargs.service_name));
    else {
        printf("%s: invalid/unknown service name or port number\r\n", pargs.service_name);
        exit(1);
    }
}
/*****************************************************************
创建守护进程函数
*****************************************************************/
void daemonize (servfd)
int servfd;
{
    int childpid, fd, fdtablesize;
    /*忽略终端 I/O 读、写和 stop 信号*/
      signal(SIGTTOU,SIG_IGN);
```




```
        signal(SIGTTIN,SIG_IGN);
        signal(SIGTSTP,SIG_IGN);
    /*通过 fork 子进程，kill 父进程，把自身转入后台*/
    if ((childpid = fork()) < 0) {
          fputs("failed to fork first child\r\n",stderr);
          exit(1);
      }
    else if (childpid > 0)
      exit(0); /*若是父进程，结束*/
      /*设为会话组长，摆脱原终端*/
    setsid(0) ;
    /*释放控制终端*/
    if ((fd = open("/dev/tty",O_RDWR)) >= 0) {
          ioctl(fd,TIOCNOTTY,NULL);
          close(fd);
     }
    /*关闭除 servfd 外的所有文件描述符*/
    for (fd = 0, fdtablesize = getdtablesize(); fd < fdtablesize; fd++)
    if (fd != servfd)
      close(fd);
      /*改变工作目录到根目录*/
      chdir("/");
      /*重设文件掩码*/
      umask(0);
}
/*****************************************************
代理处理函数
*****************************************************/
void do_proxy (usersockfd)
int usersockfd;
{
      int isosockfd;
      fd_set rdfdset;
      int connstat;
      int iolen;
      char buf[2048];
      /*作为一个客户端，新开一个端口以连接远端服务器*/
      if ((isosockfd = socket(AF_INET,SOCK_STREAM,0)) < 0)
      errorout("failed to create socket to host");
      /*发送连接请求*/
      connstat = connect(isosockfd,(struct sockaddr *) &hostaddr, sizeof(
hostaddr));
```

```
  /*出错处理*/
 switch (connstat) {
 case 0:
 break;
 case ETIMEDOUT:
 case ECONNREFUSED:
 case ENETUNREACH:
 strcpy(buf,sys_myerrlist[errno]);
 strcat(buf,"\r\n");
 write(usersockfd,buf,strlen(buf));
 close(usersockfd);
 exit(1);
 break;
 default:
 errorout("failed to connect to host");
 }
 /*现在已经建立了连接，进入代理的数据反馈循环*/
 while (1) {
      /* 使用 select 进行并发处理 */
      FD_ZERO(&rdfdset);
      FD_SET(usersockfd,&rdfdset);
      FD_SET(isosockfd,&rdfdset);
 if (select(FD_SETSIZE,&rdfdset,NULL,NULL,NULL) < 0)
   errorout("select failed");
   /*客户端有数据发过来吗*/
 if (FD_ISSET(usersockfd,&rdfdset)) {
        /*小于或等于 0 意味着客户端已断*/
    if ((iolen = recv(usersockfd,buf,sizeof(buf),0)) <= 0)
      break;
      send(isosockfd,buf,iolen,0);
      /*把数据复制一份发给服务器端*/
   }
 /*远端服务器有数据发过来吗*/
 if (FD_ISSET(isosockfd,&rdfdset)) {
     f ((iolen = recv(isosockfd,buf,sizeof(buf))) <= 0)
          break; /*接收数据长度小于或等于 0 表明连接已断*/
          send(usersockfd,buf,iolen);
          /*把数据复制一份发给客户端*/
     }
   }
   close(isosockfd);
   close(usersockfd);
```




```
    }
/***************************************************************
出错处理函数
***************************************************************/
void errorout (msg)
char *msg;
{
    FILE *console;
    console = fopen("/dev/console","a");
    fprintf(console,"proxyd: %s\r\n",msg);
    fclose(console);
    exit(1);
}
```

13.3.3 代码分析

下面对上面的代理服务器的实例代码进行说明。

1. main()函数

main()函数的总体框架和在前面介绍的TCP服务器架构相似，使用socket()建立套接口，把代理服务器的地址与端口信息绑定到这个套接口上。然后，在这个套接口上调用 listen()函数侦听，再创建守护进程，进入一个永真的循环。使用 accept()函数等待接收连接请求，如果没有，就在这里阻塞；如果有，就使用新的套接口来描述对端的信息。最后生成一个子任务，在子任务中调用 do_proxy()处理连接，处理完后关闭新生成的套接字。

2. parse_args()函数

在本例中，代理服务器的服务端口以及所代理的远端服务器的地址和端口信息都是通过命令行参数输入的。这些参数都是字符型的，对于远端服务器的地址，既可以用主机名表示，也可以用 IP 地址表示；同时远端服务器的端口号可以用数字表示，也可以用端口服务名表示，但是需要把这些参数转化为与全局变量对应的格式，并且存储到全局变量中。实现这一功能的函数是 parse_args()。

这个函数的作用是传递命令行参数，参数的传递是通过两个全局变量来实现的，这两个变量是 int proxy_port 和 struct sockaddr_in hostaddr，分别用于传递等待连接请求的 proxy 端口和被绑定的主机网络信息。在进行局部变量定义以后，函数首先要检测命令行参数是否符合程序的要求，即在命令后紧跟代理服务器端口、远程主机名和服务端口号，如果不满足上述要求，则代理服务器程序结束。如果满足上述要求，则将命令行的 3 个参数存储进函数自定义的 pargs 结构之中。

下面就要将命令行的 3 个参数变换成合适的形式赋值给全局变量 proxy_port 和 hostaddr，以供其他函数调用。首先传送代理服务器端口 pargs.proxy_port，在这里程序调用

了一个系统函数 isdigit()以检验用户输入的端口号是否有效。isdigit()函数用来检测其参数是否是数字 1-9 中间的一个，如果答案是肯定的，则返回非 0 值；反之，返回 0。在将有效端口号传递给全局变量 proxy_port 之前，还要将其转换成为按网络字节顺序排列的数字。

经过上面的工作之后，就可将命令行的另外两个参数<remote_host>和<service_port>传递给全局变量 hostaddr 的两个成员 hostaddr.sin_port 和 hostaddr.sin_addr 了。这里使用 struct hostent *hostp 和 struct servent *servp 两个局部变量来传递参数信息。

struct hostent 在 13.2 节已经进行了说明。servent 结构说明如下：

```
struct servent {
    char s_name;            /*服务的正式名称，如 ftp、http 等*/
    char **s_aliases;       /*服务的别名列表*/
    int s_port;             /*服务的端口号*/
    char *s_proto;          /*应用协议的类型*/
};
```

在这里要用 hostent 来传递期望绑定的远程主机名（或 IP 地址）。因为命令行中的主机名（或 IP 地址）参数已经被存储进 pargs.isolated_host 中，所以就调用 inet_addr()函数对主机名或主机的 IP 地址进行二进制和字节顺序转换。inet_addr()函数的描述为：

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
unsigned long int inet_addr(const char *cp)
```

inet_addr()的作用就是将参数 cp 指向的 Internet 主机地址从数字加点表示的形式转换成二进制形式并同时转换为网络字节顺序，并将转换结果直接返回。如果 cp 指向的 IP 地址不可用，则函数返回 INADDR_NONE 或 "-1"。

如果用户在命令行中输入的是远程主机的 IP 地址，那么只用 inet_addr()就算完成任务了；如果用户输入的是主机域名，那么在例程中要用如下语句进行转换：

```
if ((inaddr = inet_addr(pargs.isolated_host)) != INADDR_NONE)
    bcopy(&inaddr,&hostaddr.sin_addr,sizeof(inaddr));
else if ((hostp = gethostbyname(pargs.isolated_host)) != NULL)
    bcopy(hostp->h_addr,&hostaddr.sin_addr,hostp->h_length);
    else {
    printf("%s: unknown host\r\n",pargs.isolated_host);
    exit(1);
}
```



其中，gethostbyname()函数就是用来转换主机域名的，它的具体描述在前面曾有介绍，这里不再赘述。例程就是通过这样的方式调用 inet_addr()和 gethostbyname()，将命令行参数中的主机域名（或 IP 地址）传递给全局变量 hostaddr 的成员 sin_addr，以便代理执行函数 do_proxy()调用。

以下将介绍如何实现传递服务名或是服务端口号。

对于服务名和服务端口号的传递，这里使用结构 servent 作传递中介。例程中使用 getservbyname()函数转换命令行参数中的服务名，此函数的详细描述为：

```
#include <netdb.h>
struct servent * getservbyname(const char *servname, const char *protoname);
```

getservbyname()函数的作用就是转换指针 servname 指向的服务名为相应的整数表示的端口号，参数 protoname 表示服务使用的协议，例程中 protoname 参数的值为 TCP_PROTO，表示使用 TCP 协议，函数调用成功就返回一个 struct servent 型的指针，其中的 s_port 成员就是服务端口号。如果用户在命令中输入的是端口号而不是服务名，那么和处理代理端口信息一样，可使用下面的语句进行处理：

```
hostaddr.sin_port= htons(atoi(pargs.service_name));
```

到这里，命令行中的参数就已经全部被转换成网络通信所要求的字节顺序和数字类型，并且存储在 3 个全局变量中，只是等待 do_proxy()函数的调用。

3. Daemonize()函数

在调用 do_proxy()处理之前，例程中还创建了一个守护进程，守护进程（Daemon）是运行在后台的一种特殊进程，它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件，是一种很有用的进程。Linux 的大多数服务器都是用守护进程实现的。例如，Internet 服务器 inetd、Web 服务器 httpd 等。同时，守护进程可以完成许多系统任务，例如，作业规划进程 crond、打印进程 lpd 等。

守护进程最重要的特性是后台运行，在这一点上与 DOS 系统下的常驻内存程序 TSR 相似。其次，守护进程必须与其运行前的环境隔离开来，这些环境包括未关闭的文件描述符、控制终端、会话和进程组、工作目录以及文件创建掩模等。这些环境通常是守护进程从执行它的父进程（特别是 shell）中继承下来的。最后，守护进程的启动方式有其特殊之处，它既可以在 Linux 系统启动时从启动脚本/etc/rc.d 中启动，可以由作业规划进程 crond 启动，还可以由用户终端（通常是 shell）执行。

守护进程编程要点如下：

- 在后台运行。

为避免挂起控制终端，将 Daemon 放入后台执行。方法是在进程中调用 fork 使父进程终止，让 Daemon 在子进程中后台执行，例如本例中。

```
if ((childpid = fork()) < 0) {
        fputs("failed to fork first child\n",stderr);
        exit(1);
    }
    else if (childpid > 0)
      exit(0); /* terminate parent, continue in child */
```

- 脱离控制终端，登录会话和进程组。

在介绍相关知识之前，有必要事先介绍一下 Linux 中的进程与控制终端、登录会话和进程组之间的关系。进程属于一个进程组，进程组号（GID）就是进程组长的进程号（PID）。登录会话可以包含多个进程组，这些进程组共享一个控制终端。这个控制终端通常是创建进程的登录终端。

控制终端，登录会话和进程组通常是从父进程继承下来的，目的就是要摆脱它们，使守护进程不受它们的影响。方法是调用 setsid()使进程成为会话组长：

```
---- setsid();
```

其中，当进程是会话组长时，setsid()将调用失败，所以在此之前必须保证进程已不是会话组长。setsid()调用成功后，进程将成为新的会话组长和新的进程组长，并与原来的登录会话和进程组脱离。由于会话过程对控制终端的独占性，进程同时与控制终端脱离。

- 关闭打开的文件描述符。

进程从创建它的父进程那里继承了打开的文件描述符。如不关闭，将会浪费系统资源，造成进程所在的文件系统无法卸载以及引起无法预料的错误。在本例中除用于连接的一个套接口描述符外，都将把它们关闭。

- 改变当前工作目录。

在进程处于活动状态时，其工作目录所在的文件系统不能卸载。一般需要将工作目录改变到根目录。

- 重设文件创建掩模。

进程从创建它的父进程那里继承了文件创建掩模。它可能修改守护进程所创建的文件的存取位。为防止这一点，需要将文件创建掩模清除，可以调用 umask(0)实现。

- 处理 SIGCHLD 信号。

处理 SIGCHLD 信号并不是必需的。但对于某些进程，特别是服务器进程往往在请求到来时生成子进程处理请求。如果父进程不等待子进程结束，子进程将成为僵尸进程（zombie）从而占用系统资源。如果父进程等待子进程结束，则将增加父进程的负担，影响服务器进程的并发性能。在 Linux 下可以简单地将 SIGCHLD 信号的操作设为 SIG_IGN。

```
--- signal(SIGCHLD,SIG_IGN);
```



这样，内核在子进程结束时就不会产生僵尸进程了。

在本例的守护进程中，除了没有对僵尸子进程进行处理外，其余的都做了。另外，在守护进程中还关闭了映射终端，/dev/tty 是一个流设备，也是映射终端。调用 close()函数将终端关闭，对以下 3 个信号进行挂空处理：

```
signal(SIGTTOU,SIG_IGN);
signal(SIGTTIN,SIG_IGN);
signal(SIGTSTP,SIG_IGN);
```

这 3 个信号的含义分别是：SIGTTOU 表示后台进程写控制终端；SIGTTIN 表示后台进程读控制终端；SIGTSTP 表示终端挂起。

4. do_proxy()函数

真正连接客户主机和远端主机的操作，都是由 do_proxy()函数来完成的。在这里先将 proxy 与远端主机绑定，然后用户通过 proxy 的绑定端口与远端主机建立连接。在 main()函数中， proxy 作为服务器程序与客户主机建立连接，而在这个 do_proxy()函数中，proxy 将与远端主机的相应服务端口（由用户在命令行参数中指定）建立连接，并负责传递用户主机和远端主机之间交换的数据。

由于要和远端主机建立连接，所以看到 do_proxy()函数的前半部分实际上相当于一段标准的客户机程序，用于首先创建一个新的套接字描述符 isosockfd，然后调用函数 connect()与远端主机之间建立连接。

在例程中 switch()函数调用用于对以下 3 种出错类型加以处理：

- ETIMEDOUT：表示超时，产生此类错误的原因有很多，最常见的是服务器忙，无法应答客户机的连接请求。
- ECONNREFUSED：表示连接拒绝，即服务器端没有已经准备好的倾听套接字，或是没有对倾听套接字的状态进行监听。
- ENETUNREACH：表示网络不可到达。

do_proxy()函数的后半部分是通过 proxy 建立用户主机与远端主机之间的连接。它既有 proxy 与用户主机连接的套接字（do_proxy()函数的参数 usersockfd），又有 proxy 与远端主机连接的套接字 isosockfd。那么最简单直接的通信建立方式就是从一个套接字读，然后直接写到另一个套接字。在本例程中使用了单进程并发处理的 select()方法。

select()函数将创建一个程序所关心的文件描述符表，它的参数将在内核中为这些文件描述符设置所关心的条件，例如是否可读、是否可写以及是否异常，而且在参数中还可以设置希望等待的最大时间。在 select()成功执行时，它将返回目前已经准备好的描述符数量，同时内核可以告诉各个描述符的状态信息。如果超时，则返回 0；如果出错，则函数返回 -1，并同时设置 errno 为相应的值。

在判断出哪个套接口准备好后，就可以使用 send()和 recv()发送和接收数据了。代理服务器把从客户端接收到的数据发送到服务器端，把从服务器端接收到的数据发往客户端。



13.4 小结

本章介绍了在嵌入式 Linux 下如何实现网络编程。强大的网络支持特性和方便的可编程性是 Linux 改造为嵌入式操作系统的一大优势，也是嵌入式 Linux 操作系统的一项重大特性。

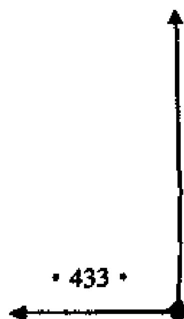
在 13.1 节中重点阐述了 Linux 的网络体系结构。首先，介绍了目前应用最为广泛的互联网络协议——TCP/IP 协议的体系结构，对 TCP/IP 协议模型中的各层的含义和作用进行了讲述。然后，介绍了嵌入式 Linux 中的 TCP/IP 结构，它的各层之间数据的封装方式、自上而下的数据流向等相关知识。

在本章 13.1 节中重点介绍了在嵌入式 Linux 中如何实现网络编程。在 Linux 中，统一向用户提供的套接口是 BSD socket，使用它可以屏蔽下层的协议差异。首先，介绍了套接字的层次结构，以及各层之间如何通过数据和操作函数指针实现连接，从而找到对应的数据和操作函数等内容。然后，对使用 BSD 套接口编程的各个 API 函数加以说明，使用这些函数的有两类协议：面向连接的 TCP 协议和无连接的 UDP 协议，并且按照服务器/客户机的模式对这两种协议进行了举例说明。最后，还介绍了如何实现 socket 编程的许多高级应用。

本章的最后是一个代理服务器的例子，这个例子对前面讲述的内容进行了总结。请认真阅读这个例程，它基本上用到了前面所讲述的所有内容，对理解 socket 编程大有好处。

13.5 思考题

1. 请画出 TCP/IP 模型，说明各层的作用。
2. 什么是套接字？它有何特性？可以分为哪些类型？
3. 说明 BSD 套接字是如何通过指针与 INET 套接字联系的。
4. 请画出 socket 编程中数据流通信的服务器端与客户端通信流程。
5. 请画出 socket 编程中数据报通信的服务器端与客户端通信流程。
6. 如何通过主机名获取 IP 地址，反过来呢？
7. 如果在服务器中要使用非阻塞等待，该如何处理？如果使用 select()处理阻塞，它的处理流程是什么？
8. 认真阅读本章所举的代理服务器的例子，画出程序流程图。



第 14 章　嵌入式数据库

知识点：

- 嵌入式数据库特点与种类
- mSQL 安装与配置
- mSQL 数据库基本操作函数

本章导读：

本章将介绍嵌入式数据库的基础知识。首先分析嵌入式系统中的数据库的特点、现状及其发展；然后重点讨论 mSQL 在嵌入式 Linux 中的应用，将详细介绍它的安装、配置以及数据库的建立过程，并结合 mSQL 提供的 API 函数讨论其数据表的操作；最后通过一个完整的实例示范如何在用户应用程序中操作 mSQL 数据库。



14.1 嵌入式系统中的数据库

同多数计算系统相似，嵌入式系统也常常需要数据库支持。虽然很多情况下可以用文件方式实现部分数据库功能，但是当应用程序需要执行一些比较复杂的数据操作（如数据排序或检索）时，文件方式就无能为力了。正因为如此，越来越多的厂商及个人开发出性能各异的嵌入式数据库产品，并且在实际应用中不断发展完善。

14.1.1 嵌入式数据库特点

由于应用环境的特殊限制，嵌入式数据库相对普通数据库系统而言有其自身特点：

- 支持常用嵌入式系统（如 Linux、Windows CE、Palm OS 等多种操作系统）和通信协议。内核小，占用内存少。
- 提供数据库功能的自由定制，能够根据具体应用或行业特点定制系统功能。
- 方便的查询功能，支持 SQL 查询语句。
- 完善的数据管理功能，支持 SQL 标准的子集，提供数据库及数据表的管理等功能。
- 操作简单方便，提供简明的 API 接口，可在高级语言中方便调用。

14.1.2 嵌入式数据库现状与发展

数据库技术发展的原动力主要来自于不断扩大的应用需求及其支撑技术的成熟。嵌入式数据库随着各种移动设备、智能计算设备、嵌入式设备的发展而迅速发展。随着嵌入式应用对数据管理的要求不断提高，嵌入式数据库技术的地位也日显重要，它将在各个应用领域中扮演越来越重要的角色。

目前国际、国内嵌入式数据库产品及其应用处于一种“百花齐放、百家争鸣”的状态，应用需求多种多样，计算平台也是各有特色，还没有任何一家厂商能够做到一统天下。

随着各种移动设备和嵌入式设备进入普通百姓的日常生活，信息共享及交流已成为人们生活中不可缺少的一部分。人们每天面对各种瞬息万变的信息资料，如果没有数据库的帮助，这一切都是不可能实现的。

此外，在未来的军事、航空、国土资源管理、移动医疗等领域嵌入式数据库系统也将占据主导作用，嵌入式数据库技术将使得信息在未来生活中无处不在、无时不在。

14.2 mSQL 简介

Mini SQL（mSQL）是嵌入式数据库家族中的佼佼者，它由澳大利亚的 David J. Hughes



开发，目前最新版本是 mSQL 3.4。本章将以 mSQL 3.0 为例对其进行相应的介绍。

Mini SQL 是一种小型的关系数据库管理系统。说它小，是因为它自身结构紧凑小巧，占用系统资源少，不像大型通用数据库那样动辄数百兆字节。事实上，mSQL 功能十分强大，足以胜任大型数据集的索引、查询任务。当然，mSQL 终究是个小型数据库系统，它的设计初衷是用于资源较少的环境下，所以某些标准 SQL 的功能它并不支持。

mSQL 的 1.x 版本只能支持有限数目记录的数据集，2.0 版本的数据库引擎已经设计为可以处理大型记录集，可以为有百万笔记录的大型记录集提供快速而一致的存取。mSQL 2.0 还包括了新的 W3-mSQL WWW 接口套件，通过使用 W3-mSQL，应用程序可将 mSQL 及其他程序结构直接植入 HTML 源代码而实现快速开发，这样就不必再为每一个具有动态内容的网页编写大量脚本。但是这些版本在功能可配置方面都做得不是很好，因此不适合在嵌入式系统中使用。

从 mSQL 3.0 版本开始，mSQL 加入了许多新特性，其中最重大的改变是它提供了两种版本的服务器端程序，以适应不同应用需求。一个是单进程的服务器 msql3d；另一个是多进程的服务器 msql3_broker。单进程版本服务器与 mSQL 2.x 的相同，而多进程版本服务器则允许多个客户端同时连上服务器，而且客户端数目可设置。MSQL 3.x 还扩大了对标准 SQL 语法的支持范围，并且它对 CPU 和内存的利用率更高。这些新特性使得 mSQL 3.x 无论是在企业级应用还是在系统资源紧张的嵌入式系统中都能轻松胜任。

14.3 在 Linux 上安装和配置 mSQL

14.3.1 mSQL 的安装

mSQL 以两种形式发布：一种是 RPM 软件包方式；另一种是用 tar 压缩的源代码方式。

RPM 软件包的安装很简单，命令如下：

rpm -ivh msql-3.0-RELEASE.i386.rpm

RPM 软件包管理器简化了系统更新的步骤，一个简单的命令就完成了所有文件的安装。

以源代码方式发布的 mSQL 的安装则要麻烦一些。首先，用 tar 程序解开压缩包：

tar –zxvf msql-3.0.tar.gz

该命令会在当前目录中建立一个新的目录 msql-3.0-RELEASE，它用于存放所有的发布文件，包括源代码目录 src 及文档目录 doc。接下来的安装步骤与一般使用 autoconf 接口的 Linux 应用程序的安装步骤有所差异，这里使用 setup 程序来设置后面的编译选项。所以，接下来应该调用命令：

./setup

该命令会将一些编译选项保存在 src/site.mm 文件中，如果用户需要改变 mSQL 程序的安装路径以及 C 编译器的类型，可以修改该文件中的对应内容。程序的默认安装路径为 /usr/local/msql3。

接下来，即可开始编译 mSQL 的源程序，执行如下命令：



make all

如果编译完成并且正确，则可以开始安装 mSQL，只需简单输入以下命令：

make install

至此，mSQL 即被正确安装在系统中，可以开始使用。

14.3.2 mSQL 系统配置

mSQL 3.0 的系统配置文件名为 msql.conf，位于安装目录下（默认为/usr/local/msql3）。另外，所有标准 mSQL 应用程序及公用程序都可以通过在执行时加上-f 参数来指定一个非标准的配置文件，以强制改变原有的默认参数值。这时当应用程序没有找到配置文件（或虽找到但有部分参数未设定），就会自动使用默认值。

1. 配置文件格式

配置文件由若干个段（section）组成，可包含空白行及注释，在注释之前应有“#”字符。每个段落都有一个段落标题，用方括号括起作为段落名称（例如：[general]）。

段落中参数值的设定方法：在参数名称之后跟上一个等号和相应的参数值，等号之后的参数值即为新值，但是每一行只能有一个参数设定项。如果配置文件中某些参数值未设定，则 mSQL 执行时会使用内部默认值。

2. 配置文件参数说明

（1）General 段。

mSQL 运行中使用的一些通用参数一般在配置文件的 general 段设定，下面分别介绍各个参数的意义。

- Inst_Dir：mSQL 安装路径，默认为/usr/local/msql3。
- DB_Dir：用户建立的数据库文件保存路径，默认为%I/msqldb。这里的%I 代表上面的 Inst_Dir，即 DB_Dir 的默认为/usr/local/msql3/msqldb。
- Msql_User：mSQL 服务器当前用户，默认值为 daemon。若有别的用户激活服务器，则系统的用户号 UID 会发生改变。
- Admin_User：特权用户，默认值为 root。特权用户可以执行特权操作，如数据库的建立与关闭等。
- Pid_File：mSQL 服务器进程号 PID 的保存文件完整路径，默认为%I/msql3d.pid。
- TCP_Port：mSQL 服务器的 TCP 服务端口，默认为 1114。基于 TCP/IP 网络的客户端通过这个端口与服务器连接。
- UNIX_Port：mSQL 服务器 UNIX 套接字文件完整路径，默认为%I/msql3.sock。本机客户端通过这个套接字文件与服务器连接。

（2）System 段。

MSQL 的系统参数在配置文件的 system 段中设定，下面分别介绍各个参数的意义。

- Msynch_Timer：定义 mSQL 服务器自动使内存数据与硬盘数据的间隔时间同步，以



秒为单位，默认值为 30。如果该值设为 0，则服务器不自动使内存与硬盘数据同步。

- Host_Lookup：决定是否需要主机 IP 地址。默认值为 True，表示不符合主机名称的连接请求将被拒绝。
- Read_Only：设置服务器工作模式为只读，拒绝任何修改数据库的操作（只接受 select 查询指令）。默认值为 False。
- Remote_Access：允许基于 TCP/IP 网络的远端用户访问 mSQL 服务器，默认值为 False。
- Local_Access：允许本机用户应用程序访问 mSQL 服务器，默认值为 True。
- Query_Log：生成日志文件，以保存服务器接收及处理的所有查询请求，默认值为 False。
- Query_Log_File：如果 Query_Log 参数被设为 True，则应设置 Query_Log_File 为该日志文件的完整路径，一般设置为%I/msql.log。
- Force_Munmap：设置 mSQL 服务器自己完成同步内存映像文件的任务，而不依靠操作系统来完成，这样做可以保证在性能不稳定的操作系统中的数据的完整性。默认值为 False。
- Num_Children：设置 mSQL 多进程服务器可以同时处理的任务数，默认值为 2。
- Table_Cache：设置数据表缓冲区中容纳的记录数，默认值为 8。这个数值越大，所占用的内存及文件描述符越多，所以在嵌入式系统中这个数值可以设得小一些，但不能小于 2。
- Sort_Max_Mem：设置执行 ORDER BY 或 DISTINCT 操作时所占用的最大内存，默认值为 1000。

3. 配置文件范例

要配置出合适的 mSQL 数据库管理系统，所要做的工作就是修改其配置文件 msql.conf。在实际应用中经常需要改动的是 Msql_User 和 Admin_User 这两个参数。Msql_User 用于设置运行 mSQL 数据库服务器程序的用户，而 Admin_User 用于设置能对 mSQL 数据库系统执行特权操作的用户。因此，如果设置 Msql_User = dbman，Admin_User = admin，则表示将由 dbman 用户运行服务器程序，由 admin 用户执行特权操作。

配置完成后，以 root 身份登录，创建 dbman 用户，命令如下：

```
useradd - g sqlusers dbman
```

接下来，将/usr/local/msql3 目录下的文件及目录的拥有者改为 dbman，执行以下命令：

```
chown - R dbman
```

再进入/usr/local/msql3/bin 目录，输入如下命令：

```
./msql3d &
```

这样就以后台执行方式启动了 mSQL 数据库系统，从而开始进行具体的数据库创建、查询等操作了。

以下是--个配置文件的简单范例，文件内容如下：

```
#
# msql.conf - Configuration file for Mini SQL 3
#
[general]
Inst_Dir = /usr/local/msql3
DB_Dir = %I/msqldb
MSQL_User = dbman
Admin_User = admin
Pid_File = %I/msql3d.pid
TCP_Port = 1114
UNIX_Port = %I/msql3.sock

[system]
Num_Children = 2
Msynch_Timer = 30
Host_Lookup = True
Read_Only = False
Remote_Access = False
Local_Access = True
Force_Mummap = False
Query_Log = False
Query_Log_File = %I/query.log
Table_Cache = 8
Sort_Max_Mem = 1000
```

14.4 mSQL 工具程序

在用户建立自己的 mSQL 数据库并在应用程序中调用之前，将首先介绍 mSQL 自身提供的工具程序及其用法，以确保整个数据库管理系统能正确运行。

mSQL 软件中包含的使用及管理数据库的工具程序有 6 个，分别用于完成数据库管理、服务器程序监视、数据库结构检索、数据转储、数据导出及导入等操作。接下来，对工具程序作分别介绍。

1. 服务器管理程序——msqladmin

msqladmin 用于对 mSQL 服务器执行管理操作，如建立新数据库、关闭服务器、数据库复制等。其调用方法为：

msqladmin [-h host] [-f confFile] [-q] Command

其中各个可选参数的意义如下：

- -h：指定服务器主机名或 IP 地址，默认值为 localhost。
- -f：指定一个非默认的配置文件，默认的配置文件是 mSQL 安装目录下的



msql.conf。

- -q：用“静态模式”执行 msqladmin。加上该参数后，msqladmin 将不会要求用户对较危险的操作（如删除一个数据库）进行确认。

Command 参数指明程序要执行的操作，可执行的操作命令如下：

- create db_name：创建一个名为 db_name 的新数据库。
- Drop db_name：删除一个名为 db_name 的数据库，同时删除该数据库中的所有内容。
- Shutdown：停止 mSQL 服务器。
- Reload：强迫服务器重新读取存取列表的 ACL 设置。
- Version：显示当前服务器的版本及配置信息。
- Stats：显示服务器的统计资料。
- Copy fromDB toDB：将 fromDB 数据库里的内容复制到 toDB 中。toDB 是一个尚未存在的数据库，如果 toDB 已经存在了，那么会返回一个错误代码。该命令通常用于进行资料备份。
- move fromDB toDB：将 fromDB 数据库改名为 toDB。

注意：大部分的系统管理操作只能以系统管理者的身份在运行服务器的主机上执行，例如，不能在远程执行关闭服务器的命令。

2. mSQL 交互程序——msql

msql 为用户提供一个与 mSQL 服务器进行对话的界面，用户可以通过它向 mSQL 服务器发送标准 SQL 命令。msql 通常是用来建立数据表或是向服务器传送 SQL 查询命令，以测试数据库内容是否正确，在用户应用程序中并不使用。使用方法如下：

msql [-h host] [-f confFile] database

参数说明如下：

- -h：指定服务器主机名或 IP 地址，默认值是 localhost。
- -f：指定一个非默认的配置文件。
- database：要进行操作的数据库名。

例如，要在数据库 mydb 中创建一个名为 mytable 的表，可以这样实现：

首先，以 mydb 为参数执行 msql 程序：

./msql mydb

程序执行后将进入 msql 对话模式，显示一个提示符 mSQL >，这时就可以输入标准 SQL 命令来建立数据表：

create table mytable(name char(10) not null,phone_number int)

然后，用\g 命令将该 SQL 语句发送给服务器执行，这样便在 mydb 数据库中建立了一个 mytable 数据表。

msql 共有 4 个控制命令，用一个反斜线加一个英文字母表示。这 4 个命令分别为：

- \q：退出 msql 程序。
- \g：将 SQL 命令发送给服务器执行。
- \e：编辑前面的 SQL 命令。



- \p：显示 SQL 命令缓存区的内容。

3. 数据库结构浏览程序——relshow

relshow 用于显示 mSQL 中数据库的结构。使用方法如下：

relshow [-h host] [-f confFile] [database [table [index]]]

参数说明如下：

- -h：指定服务器主机名或 IP 地址，默认值是 localhost。
- -f：指定一个非默认的配置文件。

如果执行 relshow 时不指定数据库名 database，则列出所有已存在的数据库的名称；如果给 relshow 指定一个数据库名称，则 relshow 会列出这个数据库中所有的数据表（table）；如果进一步指定 table 的名称，那么 relshow 会列出该数据表的结构信息（如各字段名称、类型、字段长度等）；如果详细提供数据库名称、数据表名称及其索引（index）名称，那么 relshow 将会显示所指定索引的结构、索引的数据类型以及组成索引的字段。

4. 数据库重建程序——msqldump

msqldump 用于产生一个包含了标准 SQL 命令的 ASCII 文本文件，通过这个文件可以重建数据库。这个 ASCII 文本文件包含了重建一个数据库所需的 CREATE TABLE、CREATE INDEX 以及 CREATE SEQUENCE 命令，以确保重新建立的数据库中的资料与原数据库相同。使用方法如下：

msqldump [-h host] [-f confFile] [-c] [-v] [-t] [-w WhereClause] database [table]

参数说明如下：

- -h：指定服务器主机名或 IP 地址，默认值是 localhost。
- -f：指定一个非默认的配置文件。
- -c：指定转储程序生成 INSERT 命令时，将各字段名称也一并列出。
- -v：指定在执行命令时显示较为详细的信息。
- -t：只导出数据表结构，不导出表中的数据。
- -w：过滤导出的数据，后面要跟一个标准 SQL 的 WHERE 语句。

5. 数据导出程序——msqlexport

msqlexport 程序以纯文本方式导出指定数据表中的数据，输出的文本数据可以用于其他应用程序，例如产生电子表格。

利用 msqlexport 程序可以对输出的文本数据格式进行灵活设置。用户自己可以自定义字段分隔符、数据中的分隔符替换字符、每项数据是否要用引用符号括起来以及用什么符号作为引用符号等。使用方法如下：

msqlexport [-h host] [-f confFile] [-v] [-s Char] [-q Char] [-e Char] database table

参数说明如下：

- -h：指定服务器主机名或 IP 地址，默认值是 localhost。
- -f：指定一个非默认的配置文件。

- -v：指定在执行命令时显示较为详细的信息。
- -s：用指定字符 Char 作为字段分隔符，默认为逗号。
- -q：用指定字符 Char 将每项数据括起来。
- -e：用指定字符 Char 来表示数据中出现的字段分隔符，默认值为一反斜杠（\）。

提示：程序执行结果默认输出到屏幕上显示，如果用户想将结果输出到文件中，可以在后面加上输出定向符>>。例如，执行以下命令会将结果保存到文件 backup 中：
msqlexport mydb mytable >> backup

6. 数据导入程序——msqlimport

msqlimport 程序的作用与 msqlexport 相反，是将一个纯文本文件内容导入到 mSQL 数据表。这时文本文件的每行作为数据表的一个记录。程序根据用户指定的字段分隔符将每行中的数据分为几个字段，为避免数据中的字符被当作分隔符，用户可以先用-e 参数来指定一个字符替换它。用于将各项数据括起来的引用符号也应预先去掉。msqlimport 程序的使用方法如下：

msqlimport [-h host] [-f confFile] [-v] [-s Char] [-q Char] [-e Char] database table

参数说明如下：

- -h：指定服务器主机名或 IP 地址，默认值是 localhost。
- -f：指定一个非默认的配置文件。
- -s：指定字段分隔符为 Char，默认为逗号。
- -v：指定在执行命令时显示较为详细的信息。
- -q：指定字符 Char 为数据两头的引用符号，数据导入时将之去掉。
- -e：用指定字符 Char 替换数据中出现的字段分隔符，默认值为一反斜杠（\）。

14.5 mSQL 的 C API 函数

对嵌入式系统而言，应用程序往往是通过调用 mSQL 的 API 函数来执行对特定数据库的操作。API 函数使得任何 C 语言程序都可以与 mSQL 的数据库引擎进行通信。

mSQL 的 API 函数库名称为 libsql.a，一般位于 mSQL 安装路径下的 lib 目录中，库中的函数在 msql.h 中定义。用户在编写应用程序时，应包含该头文件，该文件一般位于 mSQL 安装路径下的 include 目录，如/usr/local/msql3/include 中。另外，在对 C 程序进行编译链接时，应加上链接参数。例如，编译链接 my_sql_app.c 程序，使用如下命令：

```
cc -c -I/usr/local/msql3/include my_sql_app.c
cc -o my_sql_app my_sql_app.c -L/usr/local/msql3/lib -lmsql
```

接下来，对 mSQL 的 API 函数作简单分类介绍。

1. 查询类函数

（1）msqlConnect()函数。



该函数定义为：

```
int msqlConnect(host)
        char    *host;
```

用于建立与 mSQL 服务器的连接。参数 host 为服务器所在主机的名称或 IP 地址，当参数 host 设为 NULL 时，表示将连接本机上运行的 mSQL 服务器。若函数调用成功，则返回一个代表连接描述符的整型值，作为连接句柄供其他 API 函数使用。在使用多进程服务器的程序中，多次调用该函数将产生多个不同的连接句柄。

（2）msqlSelectDB()函数。

该函数定义为：

```
int msqlSelectDB(sock,db)
        int     sock;
        char    *db;
```

在对一个数据库进行查询等操作之前，必须先调用 msqlSelectDB()函数选中它。参数 sock 为调用 msqlConnect()函数返回的连接句柄，db 为要选择的数据库名称。当应用程序成功选中某个数据库后，就可以对该数据库进行各种操作。

（3）msqlQuery()函数。

该函数定义为：

```
int   msqlQuery(sock,q)
        int     sock;
        char    *q;
```

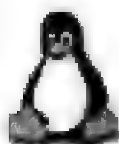
该函数用于向 mSQL 服务器提交 SQL 操作命令，例如 SELECT、DELETE、UPDATE 等 SQL 命令。参数 sock 为调用 msqlConnect()函数返回的连接句柄，q 为标准 SQL 语句。如：

```
select * from mydb ,
```

执行该函数后，从服务器返回的数据保存在内存中，应用程序需紧接着调用下面将介绍的 msqlStoreResult()函数以获得查询结果，否则一旦应用程序提交了其他查询请求，本次查询结果将丢失。

注意：mSQL 3.0 以上版本中，msqlQuery()函数返回值为服务器返回记录集数目，而 mSQL 3.0 以前的版本需用 msqlNumRows()函数来得到该值。

（4）msqlStoreResult()函数。



函数定义为：

```
m_result * msqlStoreResult()
```

msqlStoreResult()函数用于保存查询结果，该结果保存在一个 m_result 结构中，应用程序通过一个指向该结构的指针来获取查询结果。

（5）msqlFreeResult()函数。

该函数定义为：

```
void msqlFreeResult(result)
        m_result   *result;
```

该函数与 msqlStoreResult()函数相对应，当应用程序已经提取了查询结果或不再需要该结果数据时，应该调用该函数来释放内存。

（6）msqlFetchRow()函数。

该函数定义为：

```
m_row    msqlFetchRow(handle)
        m_result *handle;
```

用于提取通过 msqlStoreResult()函数保存的查询结果的某个记录数据，该函数将数据保存在一个 m_row 数组中。

（7）msqlDataSeek()函数。

函数定义为：

```
void    msqlDataSeek(handle, offset)
        m_result *handle;
        int     offset;
```

用于移动服务器响应用户查询所返回的记录集的游标。参数 handle 指向一个保存查询结果的 m_result 结构体，offset 为当前记录位置，起始值为 0。应用程序通过调用 msqlDataSeek()函数移动记录集游标后，再调用 msqlFetchRow()函数来提取相应记录。

（8）msqlFetchField()函数。

函数定义为：

```
m_field    * msqlFetchField(handle)
        m_result *handle;
```

用于获取某个查询字段的信息，包括字段名、表名、字段数据类型、字段长度以及字段属性，返回结果将保存在一个 m_field 结构中。

（9）msqlFieldSeek()函数。

函数定义为：

```
void  msqlFieldSeek(handle, offset)
      m_result *handle;
      int   offset;
```

与 msqlDataSeek()函数用法相似，该函数用于移动查询字段的游标，参数 handle 指向一个保存查询结果的 m_result 结构体，offset 为当前字段序号，起始值为 0。应用程序通过调用 msqlFieldSeek()函数移动字段游标后，再调用 msqlFetchField()函数来提取相应字段信息。

（10）msqlClose()函数。

函数定义为：

```
void msqlClose(sock)
      int   sock;
```

当应用程序不再使用某个已打开的数据库时，应及时调用 msqlClose()函数关闭连接，以释放系统资源。参数 sock 为用 msqlConnect()函数打开的连接句柄。

2. 纲要类函数

（1）msqlListDBs()函数。

该函数定义为：

```
m_result * msqlListDBs(sock)
      int   sock;
```

用于获取当前连接的 mSQL 服务器上的数据库列表。

（2）msqlListTables()函数。

该函数定义为：

```
m_result * msqlListTables(sock)
      int   sock;
```

用于查询当前选中的数据库中的数据表。

（3）msqlListIndex()函数。



该函数定义为：

```
m_result * msqlListIndex(sock,table, index)
        int     sock;
        char    *table,*index;
```

用于获得数据表索引，返回结果保存于 m_result 结构中。

3. 日期时间类函数

mSQL 的 API 函数中还包含了丰富的日期时间转换函数，以满足用户所需格式。例如，msqlTimeToUnixTime() 函数用于将 mSQL 时间格式转换为 UNIX 时间格式；msqlUnixTimeToDate()函数用于将一个 UNIX 格式的时间值转换为日期格式等。限于篇幅，这里不作一一介绍。

4. 其他类型函数

mSQL 中还有一些其他类型的函数，这里只简单介绍一下配置文件加载函数，即 msqlLoadConfigFile()函数。msqlLoadConfigFile()函数定义为：

```
int msqlLoadConfigFile(file)
    char    *file;
```

用于加载一个非默认的配置文件，参数 file 为要加载的配置文件名。

14.6 mSQL 嵌入式数据库应用实例分析

应用前面介绍的 mSQL 工具程序及 C API 函数，用户就可以建立自己的数据库并在应用程序中进行访问。一般来说，用户首先是用服务器管理程序 msqladmin 创建数据库，用监视程序 msql 建立数据表及表中的数据记录，完成数据库的创建工作，然后在 C 语言程序中调用 mSQL 的 C API 函数对数据库进行访问。

以下的 C 语言程序示范了 mSQL 的 C API 函数的使用方法。程序编译链接后，以数据库名为运行参数执行，例如：

test mydb

运行后，将在屏幕上打印出 mytable 数据表的所有客户资料，数据库已预先建立好，名为 mydb。该示范程序具体代码如下：

```
/*文件名：test.c*/
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <libmsql/msql.h>
#define VERBOSE
/*SQL 查询语句定义*/
#define SELECT_QUERY "select * from mytable"

int main(argc,argv)
        int     argc;
        char    *argv[];
{
        int     sock,   numrow,num;
        char    qbuf[160],
            host,           /*服务器所在主机名*/
            db,             /*数据库名*/
        m_result *res;
        m_row row;
        /*获取程序执行参数，即主机名和数据库名*/
        if (argc == 5) {
                        host = argv[2];
                        db = argv[3];
          }
          else {
                        host = NULL;
                        db = argv[1];
          }
        /*连接 mSQL 服务器*/
        if ((sock = msqlConnect(host)) < 0)
        {
            printf("Couldn't connect to engine!\n%s\n\n", msqlErrMsg);
            perror("");
            exit(1);
        }
        /*选择数据库*/
        if (msqlSelectDB(sock,db) < 0)
        {
            printf("Couldn't select database %s!\n%s\n",argv[1],msqlErrMsg);
        }
        /*向服务器发送 SQL 查询命令*/
        sprintf(qbuf,SELECT_QUERY);
        if(numrow=msqlQuery(sock,qbuf) < 0)
```




```
    {
        printf("Query failed (%s)\n",msqlErrMsg);
        exit(1);
    }
    /*获取查询结果*/
    res=msqlStoreResult();
    /*查询结果记录数*/
    numrow=msqlNumRows(res);
    /*移动记录集游标，获取字段信息并显示*/
    for(num=0;num<numrow;num++)
    {
        msqlDataSeek(res,num);
        row=msqlFetchRow(res);
        printf("%s,%d \n",row[0],row[1]);
    }
    /*关闭服务器连接*/
    msqlClose(sock);
    exit(0);
}
```

14.7 小结

本章以 mSQL 为例介绍了有关嵌入式数据库的相关知识，重点介绍了 mSQL 的数据库工具集的使用以及 C API 函数的定义和调用方法，全部内容围绕如何创建一个数据库并对它进行数据检索操作来展开。

嵌入式数据库是大型数据库在嵌入式系统中的延伸，它使得嵌入式小型设备也具有了大批量信息的存储与检索能力。本章介绍的只是嵌入式数据库的基本知识及普遍应用，事实上，实际中的应用远比这些要广泛和复杂得多。例如，移动手持设备中的数据库与远端服务器的数据同步、实时性能的保证等内容，都需要作更深入细致的研究。但是，万变不离其宗，相信有了这些基本知识，读者再去学习更深入的内容时就会轻松得多。

14.8 思考题

1. 到 mSQL 网站 www.hughes.com.au 下载一份源代码方式的分发软件包，在自己的机器上正确安装，并掌握配置文件的修改方法。

2. 用 mSQL 的自带工具检查软件是否正确安装，并建立一个自己的数据库。

3. 参考示范程序，编写一个 C 语言程序将自己建立的数据库中的内容读出并显示。

4. 编写一个 MiniGUI 图形界面程序，实现向自己建立的数据库中插入数据记录以及将所有记录读出显示的功能。

参 考 文 献

1. Alessandro Ruibini 著. Linux 设备驱动程序. LISOLEG 译. 北京：中国电力出版社，2000

2. 华恒公司. 华恒科技 HHPPC860-3COM-2ETH-R1 技术手册

3. 邹思轶. 嵌入式 Linux 设计与应用. 北京：清华大学出版社，2002

4. 周巍松等. Linux 系统分析与高级编程技术. 北京：机械工业出版社，1999

5. Michael K. Johnson 著. Linux 编程权威指南. 龙华乔，胡以逵译. 北京：中国电力出版社，2001

6. 毛德操. Linux 内核源代码情景分析. 杭州：浙江大学出版社，2001

7. John Lombardo 著. 嵌入式 Linux. 吴雨浓译. 北京：中国电力出版社，2003